A Survey of Pipelined Workflow Scheduling: Models and Algorithms

ANNE BENOIT
École Normale Supérieure de Lyon, France
and
ÜMİT V. ÇATALYÜREK
Department of Biomedical Informatics and Department of Electrical & Computer Engineering, The Ohio State University
and
YVES ROBERT
École Normale Supérieure de Lyon, France & University of Tennessee Knoxville
and
ERIK SAULE
Department of Biomedical Informatics, The Ohio State University

A large class of applications need to execute the same workflow on different data sets of identical size. Efficient execution of such applications necessitates intelligent distribution of the application components and tasks on a parallel machine, and the execution can be orchestrated by utilizing task-, data-, pipelined-, and/or replicated-parallelism. The scheduling problem that encompasses all of these techniques is called *pipelined workflow scheduling*, and it has been widely studied in the last decade. Multiple models and algorithms have flourished to tackle various programming paradigms, constraints, machine behaviors or optimization goals. This paper surveys the field by summing up and structuring known results and approaches.

Categories and Subject Descriptors: F.2.2 [**Nonnumerical Algorithms and Problems**]: Sequencing and scheduling; C.1.4 [**Parallel Architectures**]: Distributed architectures

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: workflow programming, filter-stream programming, scheduling, pipeline, throughput, latency, models, algorithms, distributed systems, parallel systems

Authors' Address: Anne Benoit and Yves Robert, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, 46 Allée d'Italie 69364 LYON Cedex 07, FRANCE, {Anne.Benoit|Yves.Robert}@ens-lyon.fr.
Ümit V. Çatalyürek and Erik Saule, The Ohio State University, 3190 Graves Hall — 333 W. Tenth Ave., Columbus, OH 43210, USA, {umit|esaule}@bmi.osu.edu.

1. INTRODUCTION

For large-scale applications targeted to parallel and distributed computers, finding an efficient task and communication mapping and schedule is critical to reach the best possible application performance. At the heart of the scheduling process is a graph, the *workflow*, of an application: an abstract representation that expresses the atomic computation units and their data dependencies. Hence, the application is partitioned into tasks that are linked by precedence constraints, and it is described by, usually, a directed acyclic graph (also called DAG), where the vertices are the tasks, and the edges represent the precedence constraints. In classical workflow scheduling techniques, there is a single data set to be executed, and the goal is to minimize the *latency* or *makespan*, which corresponds to the total execution time of the workflow, where each task is executed once [KA99b].

The graphical representations are not only used for parallelizing computations. In mid 70s and early 80s, a graphical representation called dataflow [Den74; Dav78; Den80] emerged as a powerful programming and architectural paradigm. Lee and Parks [LP95] present a rigorous formal foundation of dataflow languages, for which they coined the term *dataflow process networks* and presented it as a special case of Kahn process networks (KPN) [Kah74]. In KPN, a group of deterministic sequential tasks communicate through unbounded first-in, first-out channels. As a powerful paradigm that implicitely supports parallelism, dataflow networks (hence KPNs) have been used to exploit parallelism at compile time [HL97] and run time [NTS+08].

With the turn of the new millennium, Grid computing [FKT01] emerged as a global cyber-infrastructure for large-scale, integrative e-Science applications. At the core of Grid computing sit Grid workflow managers that schedule coarse-grain computations onto dynamic Grid resources. Yu and Buyya [YB05] present an excellent survey on workflow scheduling for Grid computing. Grid workflow managers, such as DAGMan [TWML01] (of the Condor project [LLM88; TTL02]), Pegasus [DShS+05], GrADS [BCC+01], Taverna [OGA+06], and ASKALON [FJP+05], utilize DAGs and *abstract workflow languages* for scheduling workflows onto dynamic Grid resources using performance modeling and prediction systems like Prophesy [TWS03], NWS [WSH99] and Teuta [FPT04]. The main focus of such Grid workflow systems are the discovery and utilization of dynamic resources that span over multiple administrative domains. It involves handling of authentication and authorization, efficient data transfers, and fault tolerance due to the dynamic nature of the systems.

The main focus of this paper is a special class of workflow scheduling that we call *pipelined workflow scheduling* (or in short *pipelined scheduling*). Indeed, we focus on the scheduling of applications that continuously operate on a stream of *data sets*, which are processed by a given wokflow, and hence the term *pipelined*. In steady-state, similar to dataflow and Kahn networks, data sets are pumped from one task to its successor. These data sets all have the same size, and they might be obtained by partitioning the input into several chunks. For instance in image analysis [SKS+09], a medical image is partitioned in tiles, and tiles are processed one after the other. Other examples of such applications include video processing [GRRL05], motion detection [KRC+99], signal processing [CLW+00; HFB+09], databases [CHM95],

molecular biology [RKO+03], medical imaging [GRR+06], and various scientific data analyses, including particle physics [DBGK03], earthquake [KGS04], weather and environmental data analyses [RKO+03].

The pipelined execution model is the core of many software and programming middlewares. It is used on different types of parallel machines such as SMP (Intel TBB [Rei07]), clusters (DataCutter [BKÇ+01], Anthill [TFG+08], Dryad [IBY+07]), Grid computing environments (Microsoft AXUM [Mic09], LONI [MGPD+08], Kepler [BML+06]), and more recently on clusters with accelerators (see for instance DataCutter [HÇR+08] and DataCutter-Lite [HÇ09]). Multiple models and algorithms have emerged to deal with various programming paradigms, hardware constraints, and scheduling objectives.

It is possible to reuse classical workflow scheduling techniques for pipelined applications, by first finding an efficient parallel execution schedule for one single data set (makespan minimization), and then executing all the data sets using the same schedule, one after the other. Although some good algorithms are known for such problems [KA99a; KA99b], the resulting performance of the system for a pipelined application may be far from the peak performance of the target parallel platform. The workflow may have a limited degree of parallelism for efficient processing of a single data set, and hence the parallel machine may not be fully utilized. Rather, for pipelined applications, we need to decide how to process multiple data sets in parallel. In other words, pipelined scheduling is dealing with both intra data set and inter data set parallelism (the different types of parallelism are described below in more details). Applications that do not allow the latter kind of parallelism are outside the scope of this survey. Such applications include those with a feedback loop such as iterative solvers. When feedback loops are present, applications are typically scheduled by software pipelining, or by cyclic scheduling techniques (also called cyclic PERT-shop scheduling, where PERT refers to Project Evaluation and Review Technique). A survey on software pipelining can be found in [AJLA95], and on cyclic scheduling in [LKdPC10].

To evaluate the performance of a schedule for a pipelined workflow, various optimization criteria are used in the literature. The most common ones are (i) the *latency* (denoted by $\mathcal{L}$), or *makespan*, which is the maximum time a data set spends in the system, and (ii) the *throughput* (denoted by $\mathcal{T}$), which is the number of data sets processed per time unit. The period of the schedule (denoted by $\mathcal{P}$) is the time elapsed between two consecutive data sets entering the system. Note that the period is the inverse of the achieved throughput, hence we will use them interchangeably. Depending on the application, a combination of multiple performance objectives may be desired. For instance, an interactive video processing application (such as SmartKiosk [KRC+99], a computerized system that interacts with multiple people using cameras) needs to be reactive while ensuring a good frame rate; these constraints call for an efficient latency/throughput trade-off. Other criteria may include reliability, resource cost, and energy consumption.

Several types of parallelism can be used to achieve good performance. If one task of the workflow produces directly or transitively the input of another task, the two tasks are said to be *dependent*; otherwise they are *independent*. *Task-parallelism* is the most well-known form of parallelism and consists in concurrently executing

independent tasks for the same data set; it can help minimize the workflow latency.

Pipelined-parallelism is used when two dependent tasks in the workflow are being executed simultaneously on different data sets. The goal is to improve the throughput of the application, possibly at the price of more communications, hence potentially a larger latency. Pipelined-parallelism was made famous by assembly lines and later reused in processors in the form of the instruction pipeline in CPUs and the graphic rendering pipeline in GPUs.

Replicated-parallelism can improve the throughput of the application, because several copies of a single task operate on different data sets concurrently. This is especially useful in situations where more computational resources than workflow tasks are available. Replicated-parallelism is possible when reordering the processing of the data sets by one task does not break the application semantics, for instance when the tasks perform a stateless transformation. A simple example of a task allowing replicated-parallelism would be computing the square root of the data set (a number), while computing the sum of the numbers processed so far would be stateful and would not allow replicated-parallelism.

Finally, *data-parallelism* may be used when some tasks contain inherent parallelism. It corresponds to using several processors to execute a single task for a single data set. It is commonly used when a task is implemented by a software library that supports parallelism on its own, or when a strongly coupled parallel execution can be performed.

Note that task-parallelism and data-parallelism are inherited from classical workflow scheduling, while pipelined-parallelism and replicated-parallelism are only found in pipelined workflow scheduling.

In a nutshell, the main contributions of this survey are the following: (i) proposing a three-tiered model of pipelined workflow scheduling problems; (ii) structuring existing work; and (iii) providing detailed explanations on schedule reconstruction techniques, which are often implicit in the literature.

The rest of this paper is organized as follows. Before going into technical details, Section 2 presents a motivating example to illustrate the various parallelism techniques, task properties, and their impact on objective functions.

The first issue when dealing with a pipelined application is to select the right model among the tremendous number of variants that exist. To solve this issue, Section 3 organizes the different characteristics that the target application can exhibit into three components: the workflow model, the system model, and the performance model. This organization helps position a given problem with respect to related work.

The second issue is to build the relevant scheduling problem from the model of the target application. There is no direct formulation going from the model to the scheduling problem, so we cannot provide a general method to derive the scheduling problem. However, in Section 4, we illustrate the main techniques on basic problems, and we show how the application model impacts the scheduling problem. The scheduling problems become more or less complicated depending upon the application requirements. As usual in optimization theory, the most basic (and sometimes unrealistic) problems can usually be solved in polynomial time, whereas the most refined and accurate models usually lead to NP-hard problems.

Although the complexity of some problems is still open, Section 4 concludes by highlighting the known frontier between polynomial and NP-complete problems.

Finally, in Section 5, we survey various techniques that can be used to solve the scheduling problem, i.e., to find the best parallel execution of the application according to the performance criteria. We provide optimal algorithms to solve the simplest problem instances in polynomial time. For the most difficult instances, we present some general heuristic methods, which aim at giving good approximate solutions.

2. MOTIVATING EXAMPLE

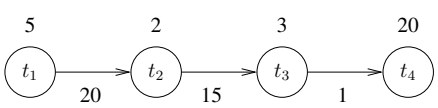
In this section, we focus on a simple pipelined application and emphasize the need for scheduling algorithms.

Consider an application composed of four tasks, whose dependencies form a linear chain: a data set must first be processed by task t_1 before it can be processed by t_2, then t_3, and finally t_4. The computation weights of tasks t_1, t_2, t_3 and t_4 (or task weights) are set respectively to 5, 2, 3, and 20, as illustrated in Fig. 1(a). If two consecutive tasks are executed on two distinct processors, then some time is required for communication, in order to transfer the intermediate result. The communication weights are set respectively to 20, 15 and 1 for communications $t_1 \rightarrow t_2$, $t_2 \rightarrow t_3$, and $t_3 \rightarrow t_4$ (see Fig. 1(a)). The communication weight along an edge corresponds to the size of the intermediate result that has to be sent from the processor in charge of executing the source task of the edge to the processor in charge of executing the sink task of the edge, whenever these two processors are different. Note that since all input data sets have the same size, the intermediate results when processing different data sets also are assumed to have identical size, even though this assumption may not be true for some applications.

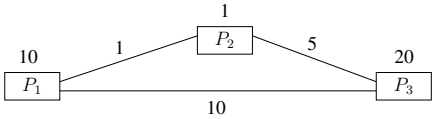
The target platform consists of three processors, with various speeds and interconnection bandwidths, as illustrated in Fig. 1(b). If task t_1 is scheduled to be executed on processor P_2, a data set is processed within $\frac{5}{1} = 5$ time units, while the execution on the faster processor P_1 requires only $\frac{5}{10} = 0.5$ time units (task weight divided by processor speed). Similarly, the communication of a data of weight c from processor P_1 to processor P_2 takes $\frac{c}{1}$ time units, while it is ten times faster to communicate from P_1 to P_3.

First examine the execution of the application when mapped sequentially on the fastest processor, P_3 (see Fig. 1(c)). For such an execution, there is no communication. The communication weights and processors that are not used are shaded in grey on the figure. On the right, the processing of the first data set (and the beginning of the second one) is illustrated. Note that because of the dependencies between tasks, this is actually the fastest way to process a single data set. The latency is computed as $\mathcal{L} = \frac{5+2+3+20}{20} = 1.5$. A new data set can be processed once the previous one is finished, hence the period $\mathcal{P} = \mathcal{L} = 1.5$.

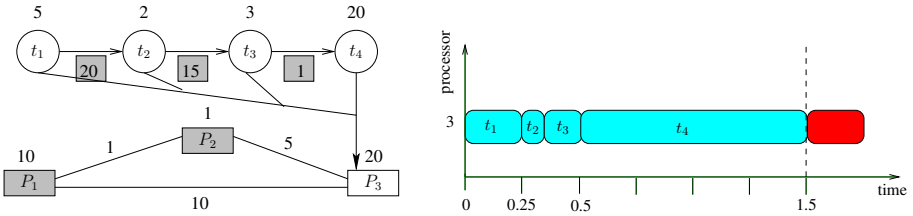
Of course, this sequential execution does not exploit any parallelism. Since there are no independent tasks in this application, we cannot use *task-parallelism* here. However, we now illustrate *pipelined-parallelism*: different tasks are scheduled on distinct processors, and thus they can be executed simultaneously on different data sets. In the execution of Fig. 1(d), all processors are used, and we greedily balance



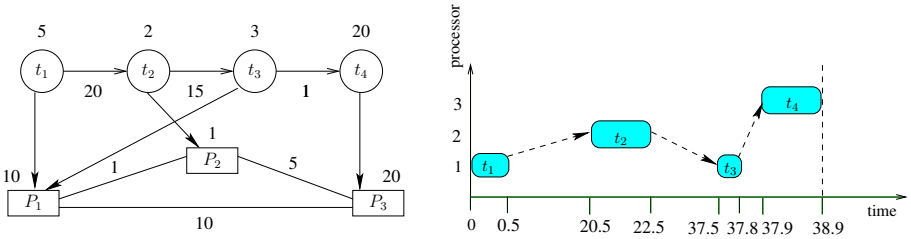
(a) Application.



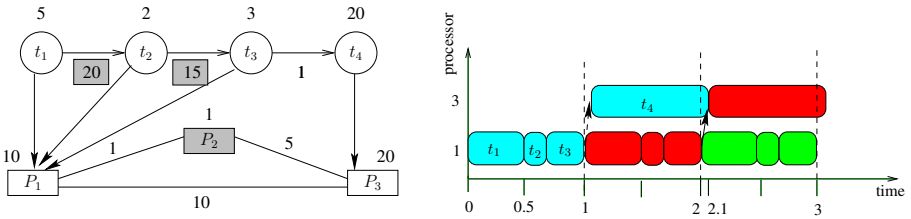
(b) Platform.



(c) Sequential execution on the fastest processor.



(d) Greedy execution using all processors.



(e) Resource selection to optimize period.

Fig. 1. Motivating example.

the computation requirement of tasks according to processor speeds. The performance of such a parallel execution turns out to be quite bad, because several large communications occur. The latency is now obtained by summing up all computation and communication times: $\mathcal{L} = \frac{5}{10} + 20 + 2 + 15 + \frac{3}{10} + \frac{1}{10} + \frac{20}{20} = 38.9$, as illustrated on the right of the figure for the first data set. Moreover, the period is not better than the one obtained with the sequential execution presented previously, because communications become the bottleneck of the execution. Indeed, the transfer from t_1 to t_2 takes 20 time units, and therefore the period cannot be better than 20: $\mathcal{P} \geq 20$. This example of execution illustrates that parallelism should be used with caution.

However, one can obtain a period better than that of the sequential execution as shown in Fig. 1(e). In this case, we enforce some resource selection: the slowest processor P_2 is discarded (in grey) since it only slows down the whole execution. We process different data sets in parallel (see the execution on the right): within one unit of time, we can concurrently process one data set by executing t_4 on P_3, and another data set by executing t_1, t_2, t_3 (sequentially) on P_1. This partially sequential execution avoids all large communication weights (in grey). The communication time corresponds only to the communication between t_3 and t_4, from P_1 to P_3, and it takes a time $\frac{1}{10}$. We assume that communication and computation can overlap when processing distinct data sets, and therefore, once the first data set has been processed (at time 1), P_1 can simultaneously communicate the data to P_3 and start computing the second data set. Finally, the period is $\mathcal{P} = 1$. Note that this improved period is obtained at the price of a higher latency: the latency has increased from 1.5 in the fully sequential execution to $\mathcal{L} = 1 + \frac{1}{10} + 1 = 2.1$ here.

This example illustrates the necessity of finding efficient trade-offs between antagonistic criteria.

3. MODELING TOOLS

This section gives general information on the various scheduling problems. It should help the reader understand the key properties of pipelined applications.

All applications of pipelined scheduling are characterized by properties from three components that we call the *workflow model*, the *system model* and the *performance model*. These components correspond to "which kind of program we are scheduling", "which parallel machine will host the program", and "what are we trying to optimize". This three-component view is similar to the three-field notation used to define classical scheduling problems [Bru07].

In the example of Section 2, the workflow model is an application with four tasks arranged as a linear chain, with computation and communication weights; the system model is a three-processor platform with speeds and bandwidths; and the performance model corresponds to the two optimization criteria, latency and period. We present in Sections 3.1, 3.2 and 3.3 the three models; then Section 3.4 classifies work in the taxonomy that has been detailed.

3.1 Workflow Model

The workflow model defines the program that is going to be executed; its components are presented in Fig. 2.

As stated in the introduction, programs are usually represented as Directed Acyclic Graphs (DAGs) in which nodes represent computation tasks, and edges represent dependencies and/or communications between tasks. The shape of the graph is a parameter. Most program DAGs are not arbitrary but instead have some predefined form. For instance, it is common to find DAGs that are a single linear chain, as in the example of Section 2. Some other frequently encountered structures are fork graphs (for reduce operations), trees (in arithmetic expression evaluation; for instance in database [HM94]), fork-join, and series-parallel graphs (commonly found when using nested parallelism [BHS+94]). The DAG is sometimes extended with two zero-weight nodes, a source node, which is made a predecessor of all entry nodes of the DAG, and a sink node, which is made a successor of all exit nodes of

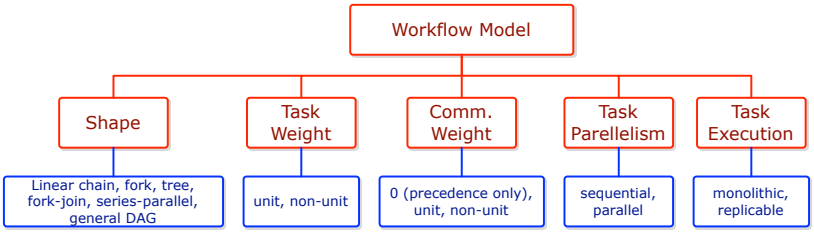


Fig. 2. The components of the workflow model.

the DAG. This construction is purely technical and allows for faster computation of dependence paths in the graph.

The weight of the tasks are important because they represent computation requirements. For some applications, all the tasks have the same computation requirement (they are said to be unit tasks). The weight of communications is defined similarly, it usually corresponds to the size of the data to be communicated from one task to another, when mapped on different processors. Note that a zero weight may be used to express a precedence between tasks, when the time to communicate can be ignored.

The tasks of the program may themselves contain parallelism. This adds a level of parallelism to the execution of the application, that is called data-parallelism. Although the standard model only uses sequential tasks, some applications feature parallel tasks. Three models of parallel tasks are commonly used (this naming was proposed by [FRS+97] and is now commonly used in job scheduling for production systems): a *rigid* task requires a given number of processors to execute; a *moldable* task can run on any number of processors, and its computation time is given by a speed-up function (that can either be arbitrary, or match a classical model such as the Amdahl's law [Amd67]); and a *malleable* task can change the number of processors it is executing on during its execution.

The task execution model indicates whether it is possible to execute concurrent replicas of a task at the same time or not. Replicating a task may not be possible due to an internal state of the task; the processing of the next data set depends upon the result of the computation of the current one. Such tasks are said to be *monolithic*; otherwise they are *replicable*. When a task is replicated, it is common to impose some constraints on the allocation of the data sets to the replicas. For instance, the dealable stage rule [Col04] forces data sets to be allocated in a round-robin fashion among the replicas. This constraint is enforced to avoid out-of-order completion and is quite useful when, say, a replicated task is followed by a monolithic one.

3.2 System Model

The system model describes the parallel machine used to run the program; its components are presented in Fig. 3 and are now described in more details.

First, processors may be identical (*homogeneous*), or instead they can have different processing capabilities (*heterogeneous*). There are two common models of *heterogeneous* processors. Either their processing capabilities are linked by a constant factor, i.e., the processors have different speeds (known as the *related* model in scheduling theory and sometimes called heterogeneous uniform), or they are not

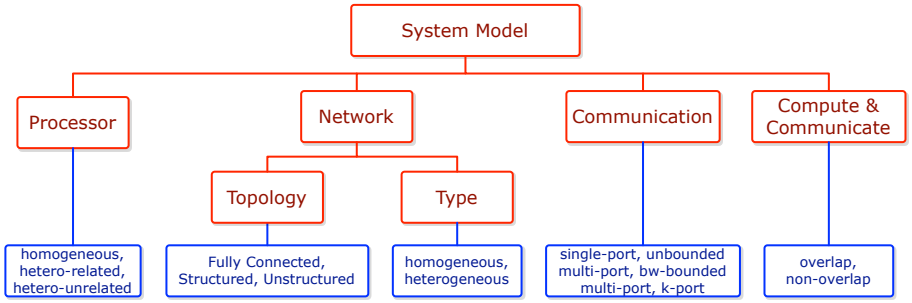


Fig. 3. The components of the system model.

speed-related, which means that a processor may be fast on a task but slow on another one (known as the *unrelated* model in scheduling theory and sometimes called completely heterogeneous). Homogeneous and related processors are common in clusters. Unrelated processors arise when dealing with dedicated hardware or from preventing certain tasks to execute on some machines (to handle licensing issues or applications that do not fit in some machine memory). This decomposition in three models is classical in the scheduling literature [Bru07].

The network defines how the processors are interconnected. The topology of the network describes the presence and capacity of the interconnection links. It is common to find *fully connected networks* in the literature, which can model buses as well as Internet connectivity. *Arbitrary networks* whose topologies are specified explicitly through an interconnection graph are also common. In between, some systems may exhibit structured networks such as *chains*, *2D-meshes*, *3D-torus*, etc. Regardless of the connectivity of the network, links may be of different types. They can be homogeneous – transport the information in the same way – or they can have different speeds. The most common heterogeneous link model is the *bandwidth* model, in which a link is characterized by its sole bandwidth. There exist other communication models such as the *delay* model [RS87], which assumes that all the communications are completely independent. Therefore, the delay model does not require communications to be scheduled on the network but only requires the processors to wait for a given amount of time when a communication is required. Frequently, the delay between two tasks scheduled on two different processors is computed based on the size of the message and the characteristics (latency and bandwidth) of the link between the processors. The *LogP* (**L**atency, **o**verhead, **g**ap and **P**rocessor) model [CKP+93] is a realistic communication model for fixed size messages. It takes into account the transfer time on the network, the latency of the network and the time required by a processor to prepare the communication. The *LogGP* model [AISS95] extends the LogP model by taking the size of the message into account using a linear model for the bandwidth. The latter two models are seldom used in pipelined scheduling.

Some assumptions must be made in order to define how communications take place. The *one-port* model [BRP03] forbids a processor to be involved in more than one communication at a time. This simple, but somewhat pessimistic, model is useful for representing single-threaded systems; it has been reported to accurately

model certain MPI implementations that serialize communications when the messages are larger than a few megabytes [SP04]. The opposite model is the *multi-port* model that allows a processor to be involved in an arbitrary number of communications simultaneously. This model is often considered to be unrealistic since some algorithms will use a large number of simultaneous communications, which induces large overheads in practice. An in-between model is the *k-port* model where the number of simultaneous communications must be bounded by a parameter of the problem [HP03]. In any case, the model can also limit the total bandwidth that a node can use at a given time (that corresponds to the capacity of its network card).

Finally, some machines have hardware dedicated to communication or use multi-threading to handle communication; thus they can compute while using the network. This leads to an *overlap* of communication and computation, as was assumed in the example of Section 2. However, some machines or software libraries are still mono-threaded, and then such an overlapping is not possible.

3.3 Performance Model

The performance model describes the goal of the scheduler and tells from two valid schedules which one is better. Its components are presented in Fig. 4.

The most common objective in pipelined scheduling is to maximize the throughput of the system, which is the number of data sets processed per time unit. In permanent applications such as interactive real time systems, it indicates the load that the system can handle. Recall that this is equivalent to minimizing the period, which is the inverse of the throughput.

Another common objective is to minimize the latency of the application, which is basically defined as the time taken by a single data set to be entirely processed. It measures the *response time* of the system to handle each data set. The objective chosen to measure response time is most of the time the maximum latency, since the latency of different data sets may be different. Latency is mainly relevant in interactive systems. Note that latency minimization corresponds to *makespan* minimization in DAG scheduling, when there is a single data set to process.

Other objectives have also been studied. When the size of the computing system increases, hardware and software become more likely to be affected by malfunctions. There are many formulations of this problem (see [BBG+09] for details), but most of the time it reduces to optimizing the probability of correct execution of the application, which is called the *reliability* of the system [GST09]. Another objective function that is extensively studied is the *energy consumption*, which has recently

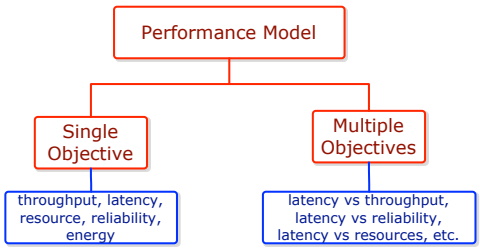


Fig. 4. The components of the performance model.

become a critical problem, both for economic and environmental reasons [Mil99]. It is often assumed that the speed of processors can be dynamically adjusted [JPG04; WvLDW10], and the slower a processor is, the less energy it consumes. Different models exist, but the main parameters are how the energy cost is computed from the speed (the energy cost is usually quadratic or cubic in the speed) and whether possible speeds are given by a continuous interval [YDS95; BKP07] or by a discrete set of values [OYI01; Pra04].

The advent of more complex systems and modern user requirements increased the interest in the optimization of several objectives at the same time. There are various ways to optimize multiple objectives [DRST09], but the most classical one is to optimize one of the objectives while ensuring a given threshold value on the other ones. Deciding which objectives are constrained, and which one remains to optimize, makes no theoretical difference [TB07]. However, there is often an objective that is a more natural candidate for optimization when designing heuristics.

3.4 Placing Related Work in the Taxonomy

The problem of scheduling pipelined linear chains, with both monolithic and replicable tasks, on homogeneous or heterogeneous platforms, has extensively been addressed in the scheduling literature [LLP98; SV96; BR08; BR10]. [LLP98] proposes a three-step mapping methodology for maximizing the throughput of applications comprising a sequence of computation stages, each one consisting of a set of identical sequential tasks. [SV96] proposes a dynamic programming solution for optimizing latency under throughput constraints for applications composed of a linear chain of data-parallel tasks. [BR08] addresses the problem of mapping pipelined linear chains on heterogeneous systems. [BR10] explores the theoretical complexity of the bi-criteria optimization of latency and throughput for chains and fork graphs of replicable and data-parallel tasks under the assumptions of linear clustering and round-robin processing of input data sets.

Other works that address specific task graph topologies include [CNNS94], which proposes a scheme for the optimal processor assignment for pipelined computations of monolithic parallel tasks with series-parallel dependencies, and focuses on minimizing latency under throughput constraints. Also, [HM94] (extended in [CHM95]) discusses throughput optimization for pipelined operator trees of query graphs that comprise sequential tasks.

Pipelined scheduling of arbitrary precedence task graphs of sequential monolithic tasks has been explored by a few researchers. In particular, [JV96] and [HO99] discuss heuristics for maximizing the throughput of directed acyclic task graphs on multiprocessor systems using point-to-point networks. [YKS03] presents an approach for resource optimization under throughput constraints. [SRM06] proposes an integrated approach to optimize throughput for task scheduling and scratch-pad memory allocation based on integer linear programming for multiprocessor system-on-chip architectures. [GRRL05] proposes a task mapping heuristic called EXPERT (EXploiting Pipeline Execution undeR Time constraints) that minimizes latency of streaming applications, while satisfying a given throughput constraint. EXPERT identifies maximal clusters of tasks that can form synchronous stages that meet the throughput constraint, and maps tasks in each cluster to the same processor so as to reduce communication overhead and minimize latency.

Pipelined scheduling algorithms for arbitrary DAGs that target heterogeneous systems include the work of [Bey01], which presents the Filter Copy Pipeline (FCP) scheduling algorithm for optimizing latency and throughput of arbitrary application DAGs on heterogeneous resources. FCP computes the number of copies of each task that is necessary to meet the aggregate production rate of its predecessors and maps these copies to processors that yield their least completion time. Later on, [SFB+02] proposed Balanced Filter Copies, which refines Filter Copy Pipeline. [BHCF95] and [RA01] address the problem of pipelined scheduling on heterogeneous systems. [RA01] uses clustering and task duplication to reduce the latency of the pipeline while ensuring a good throughput. However, these works target monolithic tasks, while [SFB+02] targets replicable tasks. Finally, [VÇK+07] (extended in [VÇK+10]) addresses the latency optimization problem under throughput constraints for arbitrary precedence task graphs of replicable tasks on homogeneous platforms.

An extensive set of papers dealing with pipelined scheduling is summed up in Table I. Each paper is listed with its characteristics. Since there are too many characteristics to present, we focus on the main ones: structure of the precedence constraints, type of computation, replication, performance metric, and communication model. The table is sorted according to the characteristics, so that searching for papers close to a given problem is made easier. Different papers with the same characteristics are merged into a single line.

The structure of the precedence constraints (the *Structure* column) can be a single chain (C), a structured graph such as a tree or series-parallel graph (S) or an arbitrary DAG (D). Processing units have computation capabilities (the *Comp.* column) that can be homogeneous (H), heterogeneous related (R) or heterogeneous unrelated (U). Replication of tasks (the *Rep.* column) can be authorized (Y) or not (N). The performance metric to compare the schedules (the *Metric* column) can be the throughput (T), the latency (L), the reliability (R), the energy consumption (E) or the number of processors used (N). The multi-objective problems are denoted with an & so that T&L denotes the bi-objective problem of optimizing both throughput and latency. Finally, the communication model (the *Comm.* column) can follow the model with only precedence constraints and zero communication weights (P), the one-port model (1), the multi-port model (M), the k-port model (k), the delay model (D) or can be abstracted in the scheduling problem (abstr). When a paper deals with several scheduling models, the variations are denoted with a slash (/). For instance, paper [BRSR08] deals with scheduling a chain (C) on either homogeneous or heterogeneous related processors (H/R) without using replication (N) to optimize latency, reliability, or both of them (L/R/L&R) under the one-port model (1).

4. FORMULATING THE SCHEDULING PROBLEM

The goal of this section is to build a mathematical formulation of the scheduling problem from a given application. As explained below, it is a common practice to consider a more restrictive formulation than strictly necessary, in order to focus on more structured schedules that are likely to perform well.

We outline some principles in Section 4.1, and then we detail a few examples to

Table I. Papers on pipelined scheduling, with characteristics of the scheduling problems.

Reference	Structure	Comp.	Rep.	Metric	Comm.
[Bok88][HNC92] [Iqb92][MO95]	C	H	N	T	P
[Nic94][PA04] [LLP98]	C	H	N	T	P
[Dev09]	C	H	N	T&L	P
[MCG+08]	C	H	Y	T	P
[BR08]	C	H/R	N	T	1
[ABR08]	C	H/R	N	T/L	M
[BRGR10]	C	H/R	N	T/L/E	1
[BRSR08]	C	H/R	N	L/R/L&R	1
[BR09]	C	H/R	Y/N	T/L/T&L	1
[BKRSR09]	C	R	N	T&L	1
[BRT09]	C	R	N	L	1
[BRSR07]	C	R	N	T/L/T&L	1
[ABMR10]	C	R	N	T/L/T&L	1/M
[dNFJG05]	C	R	Y	T&N	M
[BGGR09]	C	R	Y	T	1/M
[KN10]	C	R	Y/N	T	M
[BR10]	C/S	H/R	Y/N	T/L&T	P
[HM94] [CHM95]	S	H	N	T	M
[CNNS94]	S	H	N	T&L	P
[JV96]	D	H	N	T	M
[HO99]	D	H	N	T&L	M
[GRRL05]	D	H	N	T&L	D
[KRC+99]	D	H	N	T&L	P
[VÇK+07]	D	H	Y	T&L	M
[VÇK+10]	D	H	Y	T&L	k
[SV95]	D	H	Y/N	T	abstr
[SV96]	D	H	Y/N	T&L	abstr
[RA01]	D	H/U	N	T&L	M
[TC99]	D	R	N	T	M
[YKS03]	D	R	N	T&N	D
[Bey01][SFB+02]	D	R	Y	T	M
[BHCF95]	D	U	N	T	D
[SRM06]	D	U	N	T	M

illustrate the main techniques in Section 4.2. Finally we conclude in Section 4.3 by highlighting the known frontier between polynomial and NP-complete problems.

4.1 Compacting the Problem

One way to schedule a pipelined application is to explicitly schedule all the tasks of all the data sets, which amounts to completely unrolling the execution graph and assigning a start-up time and a processor to each task. In order to ensure that all dependencies and resource constraints are fulfilled, one must check that all predecessor relations are satisfied by the schedule, and that every processor does not execute more than one task at a given time. To do so, it may be necessary to associate a start-up time to each communication, and a fraction of the bandwidth used (multi-port model). However, the number of tasks to schedule could be extremely large, making this approach highly impractical.

To avoid this problem, a solution is to construct a more compact schedule, which hopefully has some useful properties. The overall schedule should be easily deduced

from the compact schedule in an incremental way. Checking whether the overall schedule is valid or not, and computing the performance index (e.g., throughput, latency) should be easy operations. To make an analogy with compilation, this amounts to transitioning from DAG scheduling to loop nest scheduling. In the latter framework, one considers a loop nest, i.e., a collection of several nested loops that enclose a sequence of scalar statements. Each statement is executed many times, for each value of the surrounding loop counters. Compiler techniques such as Lamport hyperplane vectors, or space-time unimodular transformations [Wol89; DRV00; KA02] can efficiently expose the parallelism within the loop nest, by providing a linear or affine closed-form expression of scheduling dates for each statement instance within each loop iteration. On the contrary, a DAG schedule would completely unroll all loops and provide an exhaustive list of scheduling dates for each statement instance.

The most common types of schedules that can be compacted are *cyclic* schedules. If a schedule has a period $\mathcal{P}$, then all computations and communications are repeated every $\mathcal{P}$ time units: two consecutive data sets are processed in exactly the same way, with a shift of $\mathcal{P}$ time units. The cyclic schedule is constructed from the *elementary* schedule, which is the detailed schedule for one single data set. If task t_i is executed on processor j at time s_i in the elementary schedule, then the execution of this task t_i for data set x will be executed at time $s_i + (x-1)\mathcal{P}$ on the same processor j in the cyclic schedule. The elementary schedule is a compact representation of the global cyclic schedule, while it is straightforward to derive the actual start-up time of each task instance, for each data set, at runtime. The relation between cyclic and elementary schedule will be exemplified in Sections 4.2.1 and 4.2.2.

With cyclic schedules, one data set starts its execution every $\mathcal{P}$ time units. Thus, the system has a throughput $\mathcal{T} = 1/\mathcal{P}$. However, the latency $\mathcal{L}$ of the application is harder to compute; in the general case, one must follow the entire processing of a given data set (but all data sets have the same latency, which helps simplify the computation). The latency $\mathcal{L}$ is the length of the elementary schedule.

Checking the validity of a cyclic schedule is easier than that of an arbitrary schedule. Intuitively, it is sufficient to check the data sets released in the last $\mathcal{L}$ units of time, in order to make sure that a processor does not execute two tasks at the same time, and that a communication link is not used twice. Technically, we can build an operation list [ABMR10] whose size is proportional to the original application precedence task graph, and does not depend upon the number of data sets that are processed.

A natural extension of cyclic schedules are *periodic* schedules, which repeat their operation every K data sets [LKdPC10]. When $K = 1$, we retrieve cyclic schedules, but larger values of K are useful to gain performance, in particular through the use of replicated parallelism. We give an example in which the throughput increases when periodic schedules are allowed. Suppose that we want to execute a single task of weight 1, and that the platform consists of three different-speed processors P_1, P_2 and P_3 with speeds 1/3, 1/5 and 1/8, respectively. For a cyclic schedule, we need to specify on which processor the task is executed, and the optimal solution is to use the fastest processor, hence leading to a throughput $\mathcal{T} = 1/3$. However, with the

use of replication, within 120 time units, P_1 can process 40 data sets, P_2 can process 24 data sets, and P_3 can process 15 data sets, resulting in a periodic schedule with $K = 40+24+15 = 79$, and a throughput $\mathcal{T} = 79/120$, about twice that of the cyclic schedule. Of course it is easy to generalize the example to derive an arbitrarily bad throughput ratio between cyclic and periodic schedules. Note however that the gain in throughput comes with a price: because of the use of replication, it may become very difficult to compute the throughput. This is because the pace of operation for the entire system is no longer dictated by a single critical resource, but instead by a cycle of dependent operations that involves several computation units and communication links (refer to [BGGR09] for details). Periodic schedules are represented in a compact way by a schedule that specifies the execution of K data sets similarly to the elementary schedule of a cyclic schedule.

Other common compact schedules consist in giving only the fraction of the time each processor spends executing each task [BLMR04; VÇK+10]. Such representations are more convenient when using linear programming tools. However, reconstructing the actual schedule involves advanced concepts from graph theory, and may be difficult to use in practice (although it can be done in polynomial time) [BLMR04].

4.2 Examples

The goal of this section is to provide examples to help the reader understand how to build a scheduling problem from the workflow model, system model and performance model. We also discuss how the problem varies when basic assumptions are modified.

4.2.1 *Chain on Identical Processors with Interval Mapping.* We consider the problem of scheduling a linear chain of n monolithic tasks onto m identical processors (with unit speed), linked by an infinitely fast network. For $1 \leq i \leq n$, task t_i has a weight p_i, and hence a processing time p_i on any processor. Fig. 5 presents an instance of this scheduling problem with four tasks of weights $p_1 = 1$, $p_2 = 2$, $p_3 = 4$, and $p_4 = 3$.

When scheduling chains of tasks, several mapping rules can be enforced:

—The *one-to-one mapping* rule ensures that each task is mapped to a different processor. This rule may be useful to deal with tasks having a high memory requirement, but all inter-task communications must then be paid.

—Another classical rule is the *interval mapping* rule, which ensures that each processor executes a set of consecutive tasks. Formally, if a processor executes tasks $t_{i_{begin}}$ and $t_{i_{end}}$, then all tasks t_i, with $i_{begin} \leq i \leq i_{end}$, are executed on the same processor. This rule, which provides an extension of one-to-one mappings, is often used to reduce the communication overhead of the schedule.

—Finally, the *general mapping* rule does not enforce any constraint, and thus any schedule is allowed. Note that for a homogeneous platform with communication costs, [ABR08] showed for the throughput objective that the optimal interval mapping is a 2-approximation of the optimal general mapping.

In this section, we consider interval mappings. Therefore, a solution to the scheduling problem is a partition of the task set $\{t_1, \ldots, t_n\}$ into m sets or in-

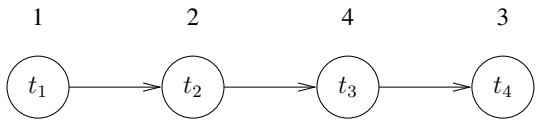


Fig. 5. An instance of the chain scheduling problem.

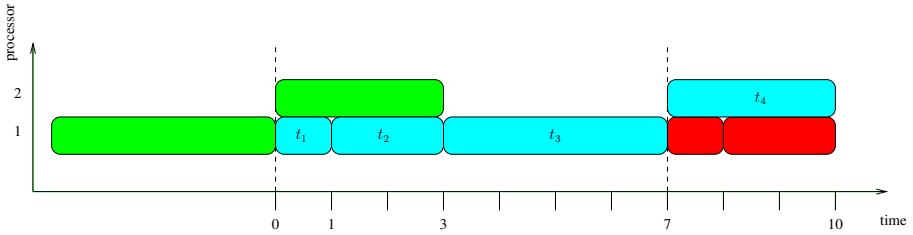


Fig. 6. The solution of optimal throughput to the instance of Fig. 5 using an interval mapping on two processors.

tervals $\{I_1, \ldots, I_m\}$, where I_j $(1 \leq j \leq m)$ is a set of consecutive tasks. Note that one could want to have fewer intervals than processors, leaving some processor(s) completely idle, but here we assume that all the processors are used to make the notation simpler. The length of an interval is defined as the sum of the processing time of its tasks: $L_j = \sum_{i \in I_j} p_i$, for $1 \leq j \leq m$. Processors are identical (with unit speed), so that all mappings of intervals onto processors are identical too.

In this case, the intervals form a compact representation of the schedule. The *elementary* schedule represents the execution of a single data set: task t_i starts its execution at time $s_i = \sum_{i' < i} p_{i'}$ on the processor in charge of its interval. An overall schedule of period $\mathcal{P} = \max_{1 \leq j \leq m} L_j$ can now be constructed: task t_i is executed at time $s_i + (x-1)\mathcal{P}$ on the x-th data set. A solution of the instance of Fig. 5 on two processors that use the intervals $\{t_1, t_2, t_3\}$ and $\{t_4\}$ is depicted in Fig. 6, where the boxes represent tasks and data sets are identified by colors. The schedule is focused on the cyan data set (the labeled tasks), which follows the green one (partially depicted) and precedes the red one (partially depicted). Each task is periodically scheduled every 7 time units (a period is depicted with dotted lines). Processor 2 is idle during 4 time units within each period.

One can check that such a schedule is valid: the precedence constraints are respected, two tasks are never scheduled on the same processor at the same time (the processor in charge of interval I_j executes tasks for one single data set during L_j time units, and the next data set arrives after $\max_{j'} L_{j'}$ time units), and the monolithic constraint is also fulfilled, since all the instances of a task are scheduled on a unique processor.

To conclude, the throughput of the schedule is $\mathcal{T} = \frac{1}{\mathcal{P}} = \frac{1}{max_{1 \leq j \leq m} L_j}$, and its latency is $\mathcal{L} = \sum_{1 \leq i \leq n} p_i$. Note that given an interval mapping, $\mathcal{T}$ is the optimal throughput since the processor for which $L_j = \max_{j'} L_{j'}$ will never be idle, and it is the one that defines the period. Note also that the latency is optimal over all schedules, since $\sum_{1 \leq i \leq n} p_i$ is a lower bound on the latency.

For such a problem (no communication, identical processors, linear dependency

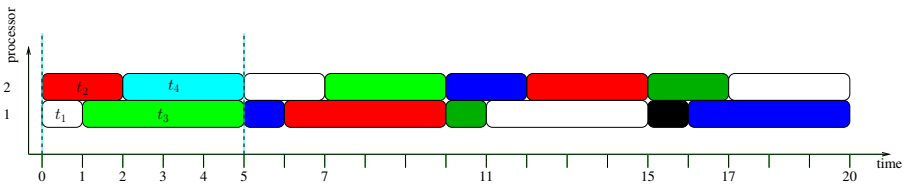


Fig. 7. The solution of optimal throughput to the instance of Fig. 5 using a general mapping on two processors.

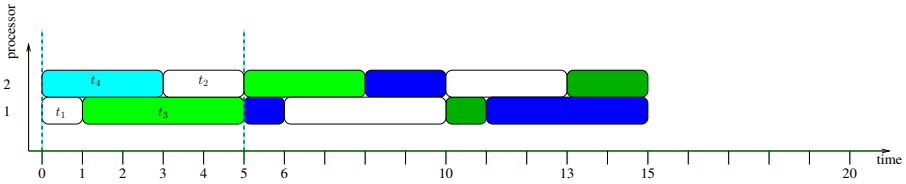


Fig. 8. A solution of the same throughput with Fig. 7, but with a better latency.

graph, no replication, interval mapping), the problem of optimizing the throughput is reduced to the classical chains-on-chains partitioning problem [PA04], and it can be solved in polynomial time, using for instance a dynamic programming algorithm.

4.2.2 *Chain on Identical Processors with General Mapping.* This problem is a slight variation of the previous one: solutions are no longer restricted to interval mapping schedules, but any mapping may be used. By suppressing the interval mapping constraint, we can usually obtain a better throughput, but the scheduling problem and schedule reconstruction become harder, as we illustrate in the following example.

The solution of a general mapping can be expressed as a partition of the task set $\{t_1, \ldots, t_n\}$ into m sets $\{A_1, \ldots, A_m\}$, but these sets are not enforced to be intervals anymore. The optimal period is then $\mathcal{P} = \max_{1 \leq j \leq m} \sum_{i \in A_j} p_i$.

We present a generic way to reconstruct from the mapping a cyclic schedule that preserves the throughput. A core schedule is constructed by scheduling all the tasks according to the allocation without leaving any idle time and, therefore, reaching the optimal period. Task t_i in set A_j is scheduled in the core schedule at time $s_i = \sum_{i' < i, i' \in A_j} p_{i'}$. A solution of the instance presented in Fig. 5 is depicted in Fig. 7 between the dotted lines (time units 0 to 5); it schedules tasks t_1 and t_3 on processor 1, and tasks t_2 and t_4 on processor 2.

The notion of core schedule is different than the notion of elementary schedule. Informally, the elementary schedule describes the execution of a single data set while the tasks in the core schedule may process different data sets.

The cyclic schedule is built so that each task takes its predecessor from the previous period: inside a period, each task is processing a different data set. We can now follow the execution of the x-th data set: it starts being executed for task t_i at time $s_i + (i + x - 1)\mathcal{P}$, as illustrated for the white data set ($x = 0$) in Fig. 7. This technique produces schedules with a large latency, between $(n-1)\mathcal{P}$ and $n\mathcal{P}$. In the example, the latency is 20, exactly 4 times the period. In Fig. 7, the core

schedule is given between the dotted lines (from time step 0 to 5). The elementary schedule is given by restricting the figure to the white data set (i.e., removing all other data sets).

The strict rule of splitting the execution in n periods ensures that no precedence constraint is violated. However, if the precedence constraint between task t_i and task t_{i+1} is respected in the core schedule, then it is possible to schedule both of them in a single time period. Consider the schedule depicted in Fig. 8. It uses the same allocation as the one in Fig. 7, but tasks t_2 and t_4 have been swapped in the core schedule. Thus, tasks t_1 and t_2 can be scheduled in the same period, leading to a latency of 13 instead of 20.

Note that the problem of finding the best general mapping for the throughput maximization problem is NP-complete: it is equivalent to the 2-PARTITION problem [GJ79] (consider an instance with two processors).

4.2.3 *Chain with a Fixed Processor Allocation.* In the previous examples, we have given hints of techniques to build the best core schedule, given a mapping and a processor allocation, in simple cases with no communication costs. In those examples, we were able to schedule tasks in order to reach the optimal throughput and/or latency.

Given a mapping and a processor allocation, obtaining a schedule that reaches the optimal latency can be done by greedily scheduling the tasks in the order of the chain. However, this may come at the price of a degradation of the throughput, since idle times may appear in the schedule. We can ensure that there will be no conflicts if the period equals the latency (only one data set in the pipeline at any time step).

If we are interested in minimizing the period, the presence of communications makes the problem much more difficult. In the model without computation and communication overlap, it is actually NP-hard to decide the order of communications (i.e., deciding the start time of each communication in the core schedule) in order to obtain the minimum period (see [ABMR10] for details). If computation and communication can be overlapped, the processor works simultaneously on various data sets, and we are able to build a conflict free schedule. When a bi-criteria objective function is considered, more difficulties arise, as the ordering of communications also becomes vital to obtain a good trade-off between latency and period minimization.

4.2.4 *Scheduling Moldable Tasks with Series-Parallel Precedence.* Chains are not the only kind of precedence constraints that are structured enough to help derive interesting results. For instance, series-parallel graphs [VTL82] are defined by composition. Given two series-parallel graphs, a series composition merges the sink of one graph with the root (or source) of the other one; a parallel composition merges the sinks of both graphs and the roots of both graphs. No other edges are added or removed. The basic series-parallel graph is composed of two vertices and one edge. Fig. 9 gives an example of a series-parallel graph. The chain of length three (given as t_2, t_5 and t_7 in Fig. 9) is obtained by composing in series the chain of length two with itself. The diamond graph (given as t_1, t_3, t_4 and t_6 in Fig. 9) is obtained by composing in parallel the chain of length three with itself.

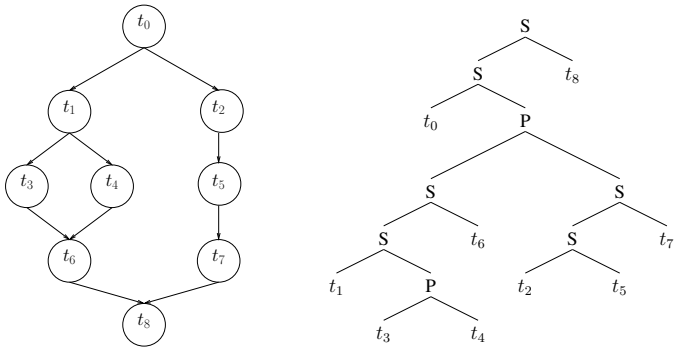


Fig. 9. A series-parallel graph, and its binary decomposition tree.

In parallel computing, series-parallel workflow graphs appear when using nested parallelism [BHS+94; BS05].

[CNNS94] considers the scheduling of series-parallel pipelined precedence task graphs, composed of moldable tasks. A given processor executes a single task and communications inside the moldable task are assumed to be included in the parallel processing times. There is no communication required between the tasks, just a precedence constraint. Provided a processor allocation, one can build an elementary schedule by scheduling the tasks as soon as possible. Since a processor is only involved in the computation of a single task, this elementary schedule reaches the optimal latency (for the processor allocation). Moreover, the elementary schedule can be executed with a period equal to the length of the longest task, leading to a cyclic schedule of optimal throughput (for the processor allocation).

Since the application task graph is a series-parallel graph, the latency and throughput of a solution can be expressed according to its Binary Decomposition Tree (BDT) [VTL82]. Each leaf of the BDT is a vertex of the graph and each internal node is either a series node $S(l, r)$ or a parallel node $P(l, r)$. A series node $S(l, r)$ indicates that the subtree l is a predecessor of the subtree r. A parallel node $P(l, r)$ indicates that both subtrees l and r are independent. A Binary Decomposition Tree is depicted in Fig. 9.

In the BDT form, the throughput of a node is the minimum of the throughputs of the children of the node: $\mathcal{T}(S(l, r)) = \mathcal{T}(P(l, r)) = \min(\mathcal{T}(l), \mathcal{T}(r))$. The expression of the latency depends on the type of the considered node. If the node is a parallel node, then the latency is the maximum of the latencies of its children: $\mathcal{L}(P(l, r)) = \max(\mathcal{L}(l), \mathcal{L}(r))$. If it is a series node, the latency is the sum of the latencies of its children: $\mathcal{L}(S(l, r)) = \mathcal{L}(l) + \mathcal{L}(r)$.

4.2.5 *Arbitrary DAGs on Homogeneous Processors.* Many applications cannot be represented by a structured graph such as a chain or a series-parallel graph. Arbitrary DAGs are more general but at the same time they are more difficult to schedule efficiently. Fig. 10 presents a sample arbitrary DAG.

Scheduling arbitrary DAGs poses problems that are similar to those encountered when scheduling chains. Consider first the case of one-to-one mappings, in which each task is allocated to a different processor. A cyclic schedule is easily built by

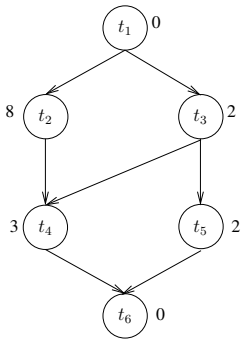


Fig. 10. An arbitrary DAG (the task weight is the label next to the task).

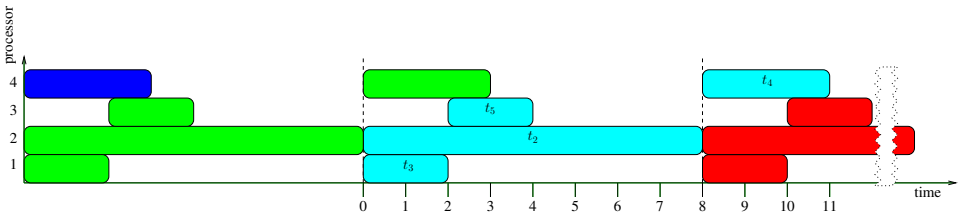


Fig. 11. One-to-one mapping of the instance of Fig. 10 with $\mathcal{L} = 11$ and $\mathcal{T} = \frac{1}{8}$. Tasks t_1 and t_6 have computation time 0, therefore they are omitted.

scheduling all tasks as soon as possible. Task i is scheduled in the cyclic schedule on processor i at time $s_i = \max_{i' \in pred(i)} s_{i'} + p_{i'}$. This schedule can be executed periodically every $\mathcal{P} = \max_i p_i$ with throughput $\mathcal{T} = \frac{1}{\max_i p_i}$. The latency is the longest path in the graph, i.e., $\mathcal{L} = \max s_i + p_i$. A schedule built in such a way does not schedule two tasks on the same processor at the same time; indeed, each task is executed during each period on its own processor, and its processing time is smaller or equal to the period. Under the one-to-one mapping constraint, this schedule is optimal for both objective functions. The solution for the graph of Fig. 10 is presented in Fig. 11, with a latency $\mathcal{L} = 11$ and a throughput $\mathcal{T} = \frac{1}{8}$.

When there is no constraint enforced on the mapping rule, problems similar to those of general mappings for linear chains appear (see Section 4.2.2): we cannot easily derive an efficient cyclic schedule from the processor allocation. Establishing a cyclic schedule that reaches the optimal throughput given a processor allocation is easy without communication cost, but it can lead to a large latency. Similarly to the case of chains, a core schedule is obtained by scheduling all the tasks consecutively without taking care of the dependencies. This way, we obtain the optimal period (for this allocation) equal to the load of the most loaded processor. The cyclic schedule is built so that each task takes its data from the execution of its predecessors in the last period. Therefore, executing a data set takes as many periods as the depth of the precedence task graph. On the instance of Fig. 10, the optimal throughput on two processors is obtained by scheduling t_2 alone on a processor. Fig. 12 presents a cyclic schedule for this processor allocation according to

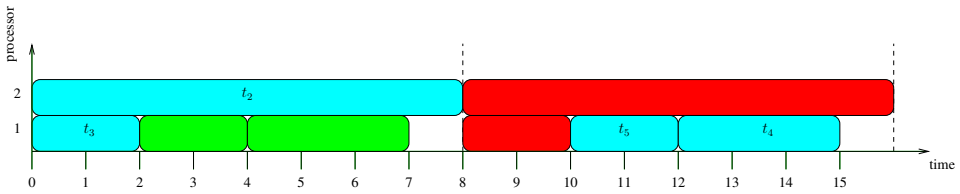


Fig. 12. A general mapping solution of the instance of Fig. 10 with $\mathcal{L} = 15$ and $\mathcal{T} = \frac{1}{8}$. Tasks t_1 and t_6 have computation time 0, therefore they are omitted.

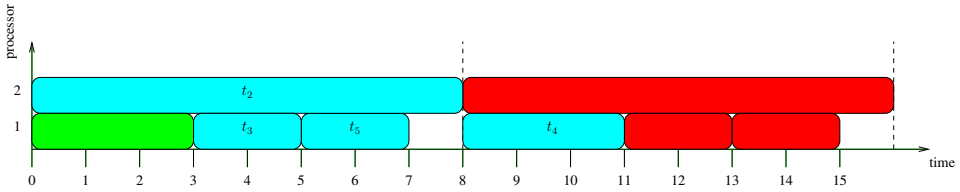


Fig. 13. A general mapping solution of the instance of Fig. 10 with $\mathcal{L} = 11$ and $\mathcal{T} = \frac{1}{8}$. Tasks t_1 and t_6 have computation time 0, therefore they are omitted.

this generic technique, leading to a latency $\mathcal{L} = 15$. Note that t_5 could be scheduled in the same period as t_3 and in general this optimization can be done by a greedy algorithm. However, it does not guarantee to obtain the schedule with the optimal latency, which is presented in Fig. 13 and has a latency $\mathcal{L} = 11$. Indeed, contrarily to linear pipelines, given a processor allocation, obtaining the cyclic schedule that minimizes the latency is NP-hard [RSBJ95].

The notion of interval mapping cannot be directly applied on a complex DAG. However, we believe that the interest of interval mapping schedules of chains can be transposed to *convex clustered schedules* on DAGs. In a convex clustered schedule, if two tasks are executed on one processor, then all the tasks on all the paths between these two tasks are scheduled on the same processor [LT02]. Convex clustered schedules are also called *processor ordered schedules*, because the graph of the inter-processor communications induced by such schedules is acyclic [GMS04]. The execution of the tasks of a given processor can be serialized and executed without any idle time (provided their execution starts after all the data have arrived). This leads to a reconstruction technique similar to the one applied on chains of tasks following the interval mapping rule. Two processors that are independent in the inter-processor communication graph can execute their tasks on a given data set during the same period in any order, without violation of the precedence constraints. Such a construction leads to the optimal throughput for a given convex clustered mapping of the tasks to the processors, and to a latency $\mathcal{L} \leq x\mathcal{P} \leq m\mathcal{P}$, where x is the length of the longest chain in the graph of communications between processors.

Algorithms to generate convex clustered schedules based on recursive decomposition have been proposed for classical DAG scheduling problems [PST05]. In pipelined scheduling, heuristic algorithms based on stages often generate convex clustered schedule such as [BHCF95; GRRL05]. However the theoretical properties of such schedules have never been studied for pipelined workflows.

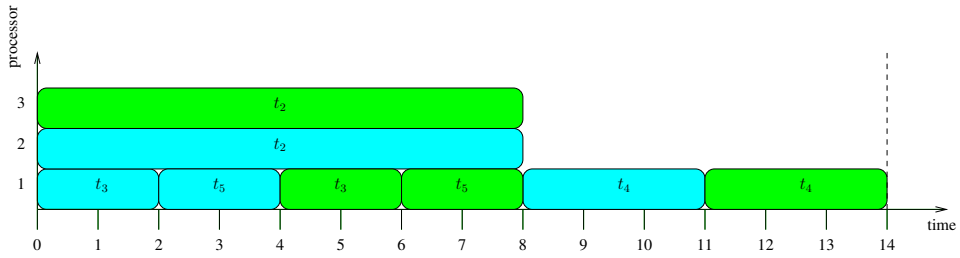


Fig. 14. A general mapping solution of the instance of Fig. 10 with $\mathcal{L} = 14$ and $\mathcal{T} = \frac{1}{7}$ when task t_2 is replicable. Tasks t_1 and t_6 have computation time 0, therefore they are omitted.

4.2.6 *Scheduling Arbitrary DAGs on Homogeneous Processors with Replication.* A task is replicable if it does not contain an internal state. It means that the same task can be executed at the same time on different data sets. Replication allows one to increase the throughput of the application. (We point out that this is different from duplication, which consists in executing the same task on the same data set on multiple different processors. Redundantly executing some operations aims at either reducing communication bottlenecks, or increasing reliability.) On the instance presented in Fig. 10, only two processors can be useful: the dependencies prevent any three tasks from being executed simultaneously, so a third processor would improve neither the throughput nor the latency for monolithic tasks. However, if task t_2 is replicable, the third processor could be used to replicate the computation of this task, therefore leading to the schedule depicted in Fig. 14.

Replicating t_2 leads to a periodic schedule that executes two data sets every 14 time units ($K = 2$). Its throughput is therefore $\mathcal{T} = \frac{2}{14} = \frac{1}{7}$, which is better than without replication. The latency is the maximum time a data set spends in the system. Without replication, all the data sets spend the same time in the system. With replication, this statement no longer holds. In the example, the cyan data set spends 11 time units in the system whereas the green one spends 14 time units. The latency of the schedule is therefore $\mathcal{L} = 14$. If task t_4 was replicable as well, two copies could be executed in parallel, improving the throughput to $\mathcal{T} = \frac{2}{11}$ and the latency to $\mathcal{L} = 11$. A fourth processor could be used to pipeline the execution of t_4 and reach a period of $\mathcal{P} = 8$ and, hence, a throughput of $\mathcal{T} = \frac{1}{4}$.

A schedule with replication is no longer *cyclic* but instead is *periodic*, with the definitions of Section 4.1. Such a schedule can be seen as a pipelined execution of an unrolled version of the graph. The overall schedule should be specified by giving a periodic schedule of length ℓ (the time between the start of the first task of the first data set of the period and the completion of the last task of the last data set of the period), detailing how to execute K consecutive data sets, and providing its period $\mathcal{P}$. Verifying that the schedule is valid is done in the same way as for classical elementary schedules: one needs to expand all the periods that have a task running during the schedule, that is to say the tasks that start during the elementary schedule and within the ℓ time units before. Such a schedule has a throughput of $\mathcal{T} = \frac{K}{\mathcal{P}}$, and its latency should be computed as the maximum latency of the data sets in the elementary schedule.

Note that if all tasks are replicable, the whole task graph can be replicated on

all the m processors. Each processor executes sequentially exactly one copy of the application. This leads to a schedule of latency and period $\mathcal{P} = \mathcal{L} = \sum_i p_i$, and a throughput of $\mathcal{T} = \frac{m}{\sum_i p_i}$.

When using replication, it is possible that data set i is processed before its predecessor $i-1$. This behavior mainly appears when processors are heterogeneous. The semantic of the application might not allow data sets to be processed in such an *out-of-order* fashion. For instance, if a task is responsible for compressing its input, providing the data sets out-of-order will change the output of the program. One can then either impose a delay, or use some other constraints, as for instance the dealable stage constraint [Col04].

4.2.7 *Model Variations.* In most cases, heterogeneity does not drastically change the scheduling model. However, the compact schedule description must then contain the processor allocation, i.e., it must specify which task is executed on which processor. Otherwise the formulations stay similar.

A technique to reduce latency is to consider *duplication* [AK98; VÇK+10]. Duplicating a task consists in executing the same task more than once on different processors for every data set. Each task receives its data from one of the duplicates of each of its predecessors. Hence, this allows more flexibility for dealing with data dependency. The idea is to reduce the communication overheads at the expense of increasing the computation load. Another goal is to increase the reliability of the system: whenever the execution of one duplicate would fail, that of another duplicate could still be successful. The major difference of duplication as compared to replication is the following: with duplication, a single data set is executed in each period, whereas with replication, several data sets can be executed in each period.

Communication models affect the schedule formulation. The easiest communication model is the one-port model where a machine communicates with a single other machine at a time. Therefore, in the schedule, each machine is represented by two processors, one for the computations and one for the communications. A valid schedule needs to "execute" a communication task at the same time on the communication processor of both machines involved in the data transfer. A common variation on the one-port model is to forbid communication and computation overlap. This model is used in [HO99]. In this case, there is no need for a communication processor; the communication tasks have to be scheduled on the computation processor [BRSR07].

To deal with more than one communication at a time, a realistic model would be to split the bandwidth equally among the communications. However such models are more complicated to analyze, and are therefore not used in practice. Two ways of overcoming the problem exist. The first one is to consider the k-port model where each machine has a bandwidth B divided equally into k channels. The scheduling problem amounts to using k communication processors per machine. This model has been used in [VÇK+10].

When only the throughput matters (and not the latency), it is enough to ensure that no network link is overloaded. One can reconstruct a periodic schedule explicitly, by using the model detailed previously, considering each network link as a processor. This approach has been used in [TC99].

4.3 Complexity

The goal of this section is to provide reference pointers for the complexity of the pipelined scheduling problem. Lots of works are dedicated to highlighting the frontier between polynomial and NP-hard optimization problems in pipelined scheduling.

The complexity of classical scheduling problems have been studied in [Bru07]. One of the main contributions was to determine some constraint changes that always make the problem harder. Some similar results are valid on pipelined scheduling. For instance, heterogeneous versions of problems are always harder than their homogeneous counterpart, since homogeneous cases can be easily represented as heterogeneous problem instances but not vice versa. A problem with an arbitrary task graph or architecture graph is always harder than the structured counterpart, and in general considering a superset of graphs makes problems harder. Also, removing communications makes the problem easier.

As seen in the previous examples, throughput optimization is always NP-hard for general mappings, but polynomial instances can be found for interval mappings. The communication model plays a key role in complexity. The optimization of latency is usually equivalent to the optimization of the makespan in classical DAG scheduling [KA99b].

The complexity of multi-objective problem instances relates to three different types of questions. First, the decision problems of multi-objective problems are directly related to those for mono-objective problems. A threshold value is given for all the objectives, and the problem is to decide whether a solution exists, that matches all these thresholds. Multi-objective decision problems are obviously harder than their mono-objective counter part. Second, the counting problem consists in computing how many Pareto-optimal solutions a multi-objective problem has; a Pareto-optimal solution is such that no other solution is strictly better than it. Finally, the enumeration problem consists in enumerating all the Pareto-optimal solution of an instance. The enumeration problem is obviously harder than the decision problem and the counting problem, since it is possible to count the number of solutions with an enumeration algorithm, and to decide whether given thresholds are feasible. A complete discussion of these problems can be found in [TB07].

The complexity class of enumeration problems expresses the complexity of the problem as a function of both the size of the input of the problem and the number of Pareto-optimal solutions leading to classes EP (Enumeration Polynomial) and ENP (for Enumeration Non-deterministic Polynomial) [TBE07]. Therefore, the decision version of a multi-objective problem might be NP-complete, but since it has an exponential number of Pareto optimal solution, its enumeration version is in EP (the problem $1 \mid\mid \sum C_i^A; \sum C_i^B$ of [AMPP04] is one of the many examples that exhibit this property). Therefore, the approaches based on exhaustive enumeration can take a long time. However, [PY00] shows that most multi-objective problems admit an approximate set of Pareto optimal solutions whose cardinality is polynomial in the size of the instance (but it is exponential in the number of objectives and in the quality of the approximation). It was also shown in [PY00] that an approximation algorithm for the decision problem can be used to derive an approximation of the Pareto set in polynomial time. These results motivate the investigation of

Table II. Summary of complexity results for period minimization of a linear task graph.

Mapping rule	**Platform type**		
	Fully Hom.	***Comm. Hom.***	***Hetero.***
one-to-one	polynomial	polynomial, *NP-hard (rep.)*	NP-hard
interval	polynomial	NP-hard	NP-hard
general	NP-hard, *polynomial (rep.)*	NP-hard	

Table III. Summary of complexity results for latency minimization of a linear task graph.

Mapping rule	**Platform type**		
	Fully Hom.	***Comm. Hom.***	***Hetero.***
one-to-one	polynomial [BR09]		NP-hard [BRSR08]
interval	polynomial [BR09]		NP-hard [BRT09]
general	polynomial [BRSR08]		

algorithms that enforce thresholds on some objectives, and optimize the other ones.

The complexity of linear graph problems has been widely studied since it roots the general DAG case, and most of the structured graph ones [BR08; BR10; BRSR07; ABR08; BRSR08; BRT09; BR09; ABMR10]. The large number of variants for these scheduling problems makes complexity results very difficult to apprehend. An exhaustive list of such results can be found in [Ben09]. We provide in Tables II and III a summary of complexity results for period and latency optimization problems, which hold for all communication models. *Fully Hom.* platforms refer to homogeneous computations and communications. *Comm. Hom.* platforms add one level of heterogeneity (heterogeneous related processors). Finally, *Hetero.* platforms are fully heterogeneous (heterogeneous related processors and heterogeneous communication links).

For the period minimization problem, the reader can refer to [BR08] for the variant with no replication, and to [BR10] otherwise (results denoted with *(rep.)*). For the latency minimization problem, we report here results with no data-parallelism; otherwise the problem becomes NP-hard as soon as processors have different speeds (related model), with no communication costs [BR10].

5. SOLVING PROBLEMS

The goal of this section is to give methods to solve the pipelined scheduling problem using exact algorithms or heuristic techniques.

5.1 Scheduling a Chain of Tasks with Interval Mappings

The first problem that we consider has been presented in Section 4.2.1. It consists in scheduling a chain of n tasks onto m identical (homogeneous) processors, without communication, and enforcing the interval mapping constraint. Section 4.2.1 states that the latency of such schedules is constant, however the throughput can be optimized by minimizing the length of the longest interval.

The optimization of the throughput problem is the same combinatorial problem as the well-known chains-on-chains partitioning problem, which has been solved by a polynomial algorithm in [Bok88], and then refined to reach lower complexity

in [Iqb92; Nic94; MO95]. For very large problems, some heuristics have also been designed to reduce the scheduling times even further (see [PA04] for a survey). The first algorithm was based on a shortest path algorithm in an assignment graph. The approach below has a lower complexity, and is easier to understand.

The core of the technique is the Probe function that takes as a parameter the processing time of the tasks and the length of the longest interval $\mathcal{P}$. It constructs intervals $I = \{I_1, \ldots, I_m\}$ such that $\max_{1 \leq j \leq m} L_j \leq \mathcal{P}$, or shows that no such intervals exist (remember that $L_j = \sum_{i \in I_j} p_i$, where p_i is the processing time of task t_i). Probe recursively allocates the first x tasks of the chain to the first processor so that $\sum_{i \leq x} p_i \leq \mathcal{P}$ and $\sum_{i \leq x+1} p_i > \mathcal{P}$ until no task remains, and then returns the schedule. If the number of intervals is less than the number of processors, this function builds a schedule having no interval of length exceeding $\mathcal{P}$. Otherwise, no schedule of maximal interval length less than $\mathcal{P}$ exists with m processors. It can be easily shown that the schedules constructed are dominant, i.e., if a schedule exists, then there is one respecting this construction. The last problem is to choose the optimal value for the threshold $\mathcal{P}$. The optimal value is obtained by using a binary search on the possible values of $\mathcal{P}$, which are tested using the Probe function. This construction is polynomial but has a quite high complexity. It is possible to reduce the complexity of the Probe function using prefix sum arrays and binary search so that fewer values of $\mathcal{P}$ can be tested by analyzing the processing time values. In the general case, the lowest complexity is reached by using Nicol's algorithm [Nic94] with the algorithm for the Probe function described in [HNC92], leading to a total complexity of $O(n + m^2 \log(n) \log(n/m))$ (see [PA04] for details).

The same idea can be used to deal with different problems, for instance with non-overlapping communications following the one-port model [Iqb92]. Since there is no overlapping, the communication time is included in the computation of the throughput of both processors involved in the communication. Then, setting the boundary of the interval on a communication larger than the length of the previous task plus its in-bound communication is never efficient: the two tasks can be merged without losing optimality. (The same argument applies towards the next task.) Detecting these tasks and merging them can be performed in linear time. The resulting chain is said to be monotonic. Then, an algorithm similar to the partitioning of a chain in intervals without communication can be applied, leading to the optimal solution [Iqb92].

The same algorithm can also be used to optimally solve the case with related processor heterogeneity (processor speeds differ) if the order in which a data set goes through the processors is known. This is the case on dedicated hardware where the processor network forces the order of execution between the processors. However, if this order is not known, the problem is NP-complete in the strong sense [BRSR07], even without taking communication costs into account. There are too many permutations to try, but the Probe algorithm sets a solid ground to build heuristics upon.

[BR08] proposes three heuristics to build interval mappings for optimizing the throughput on heterogeneous processors. The first one, called SPL, starts by assigning all the tasks to the fastest processor and then greedily splits the largest interval by unloading work to the fastest available processor. The splitting point is chosen

so as to minimize the period of the new solution. The two other heuristics BSL and BSC use a binary search on the period of the solution. This period is used as a goal in the greedy allocation of the tasks to the processors. BSL allocates the beginning of the chain of tasks to the processor that will execute the most computations while respecting the threshold. On the other hand, BSC chooses the allocation that is the closest to the period. Note that [PTA08] surveys chains-on-chains partitioning problems with heterogeneous resources, and proposes several heuristics that can be transposed to throughput optimization without communication costs.

[KN10] proposes a heuristic algorithm called ThroughputPipeline to schedule a chain using interval mappings on Grid computing systems (related processors, bounded multi-port, communication and computation overlapping, no replication), considering routing through intermediate nodes. ThroughputPipeline is based on Dijkstra's shortest path algorithm. Simulation shows that its performance is within 30% of the general mapping optimal solution (obtained by an Integer Linear Programming formulation).

Solving the problem of optimizing both the latency and the throughput of a linear pipeline application has been considered in [SV96; BRSR07; BKRSR09]. Bi-objective optimization problems are usually solved by providing a set of efficient solutions. This set of solutions is generated by using an algorithm that targets values of one objective while optimizing the other one. The solution space is explored by executing this algorithm with different values of the threshold, hence covering the efficient part of the solution space.

[SV95] addresses the problem of scheduling a chain of tasks on homogeneous processors to optimize the throughput of the application without computation and communication overlap. It covers a large range of problems since it addresses moldable tasks with dedicated communication functions and replicable tasks. The network is supposed to be homogeneous but the details of the communication model are abstracted by explicitly giving the communication time in the instance of the problem. The technique used is based on dynamic programming and leads to an optimal polynomial algorithm. This result can be extended by adding a latency dimension in the dynamic program to allow the optimization of the latency under throughput constraint [SV96].

[BRSR07] proposes heuristics that optimize the throughput and latency when link bandwidths are identical but processors have different speeds (one-port communications without overlap). Six heuristics are presented, enforcing a constraint on either the throughput or the latency. All six heuristics are similar to SPL in that they start by allocating all the tasks to the fastest processor and split the interval iteratively. The differences are that each interval may be split in two to use the fastest available processor, or split in three to use the fastest two processors available. The other differences are about the solution chosen; it could be the one that maximizes one objective or a ratio of improvement. [BKRSR09] proposes an integer linear programming formulation to solve the problem optimally (and with heterogeneous bandwidths). The solving procedure takes a long time even on a simple instance of 7 tasks and 10 processors (a few hours on a modern computer) but allows assessment of the absolute performance of the previously proposed heuristics.

5.2 Scheduling a Chain of Tasks with General Mappings

Using general mappings instead of restricting to interval mappings leads to better throughput. Without replication and without communication costs, the optimization of the throughput on homogeneous processors is NP-complete by reduction to 3-PARTITION. In fact, the mathematical problem is to partition n integers $p_1, \ldots, p_n$ into m sets $A_1, \ldots, A_m$ so that the length of the largest set $\max_{1 \leq j \leq m} \sum_{i \in A_j} p_i$ is minimized. This formulation corresponds exactly to the problem of scheduling independent tasks on identical processors to minimize the makespan, that has been studied for a long time [Gra66; Gra69], and considered theoretically solved since [HS87].

On homogeneous processors, the classical List Scheduling algorithm schedules tasks greedily on the least loaded processor, and it is a 2-approximation [Gra66], i.e., the value of the obtained solution is at most twice the optimal value. Sorting tasks by non increasing processing times leads to the Largest Processing Time (LPT) algorithm, which is known to be a 4/3-approximation algorithm [Gra69]. An approximation algorithm with arbitrary precision that is polynomial in the size of the problem but exponential in the inverse of the precision (also known as PTAS) based on binary search and dynamic programming has been proposed in [HS87]. The question of whether an algorithm with arbitrary precision that is polynomial in the size of the problem and in the inverse of the precision (also known as FPTAS) arises. However, the problem is NP-complete in the strong sense and FPTAS do not exist for this problem unless P=NP (see [Hoc97] for details on complexity classes based on approximation properties).

With heterogeneous processors, there is still a link with the classical makespan optimization problem. If processors are heterogeneous related (computing at different speeds), the throughput optimization problem is the same as scheduling independent tasks on heterogeneous related processors. This problem admits a 2-approximation algorithm similar to LPT [GIS77]. [HS88] provides an elaborate approximation scheme with very high runtime complexity as well as a simple 3/2-approximation algorithm. If processors are unrelated (i.e., their speeds depend on the task they are handling), the throughput optimization problem is the same as scheduling independent tasks on unrelated processors to minimize the makespan. It can be shown that there exists no approximation algorithm with a ratio better than 3/2 [LST90]. Moreover, a 2-approximation algorithm based on binary search and linear programming has been proposed in [LST90] and recently made simpler and faster by [GMW05].

The results on classical scheduling problems remain valid even if the graph is not linear as long as the performance index is the throughput and there are no communications. However, they still provide an interesting baseline to assess the impact of communications.

[KN10] considers the problem of scheduling a chain on a grid computer with routing to optimize the throughput. Processors are heterogeneous (related) and communications follow the bounded multi-port model with overlapping. It first considers the case with replication (called multi-path in this terminology). An optimal general mapping is constructed in polynomial time by LPSAG, a flow-based linear programming formulation (somehow similar to LPsched [dNFJG05]). Finding

the optimal general mapping without replication (single-path) is investigated and shown to be NP-complete. LPSAG is extended into an Integer Linear Program that finds the optimal general mapping without replication (but possibly in exponential time). A polynomial heuristic based on Dijkstra's shortest path algorithm that only constructs interval mappings is proposed. Experiments show that the heuristic is within 30% of the Integer Linear Programming solution.

[BRT09] provides a polynomial algorithm to optimize the latency of a pipelined chain on heterogeneous (related) networks of processors under the one-port model. The algorithm is based on a dynamic programming formulation.

5.3 Structured Application Graphs

[CNNS94] tackles the problem of scheduling pipelined series-parallel graphs of moldable tasks. (This problem has been presented in Section 4.2.4.) A bi-criteria problem is solved optimally by optimizing the latency under throughput constraint, under the assumption that a processor executes at most one task. The latency is optimized by computing, for each node of the binary decomposition tree, the optimal latency achievable for any number of processors using dynamic programming. The latency of the series node $S(l,r)$ using m processors is obtained by evaluating $\mathcal{L}(S(l,r),m) = \min_{1\leq j\leq m-1}(\mathcal{L}(l,j)+\mathcal{L}(r,m-j))$. The latency of the parallel node $P(l,r)$ on m processors is obtained by evaluating $\mathcal{L}(P(l,r),m) = \min_{1\leq j\leq m-1}\max(\mathcal{L}(l,j),\mathcal{L}(r,m-j))$. The leaves of the binary decomposition tree are the tasks of the application and their latency are given as an input of the problem. The throughput constraint is ensured by setting the latency of the tasks to infinity on processor allocations that would not respect the throughput constraint. Evaluating the latency of a node for a given number of processors requires $O(m)$ computations and there are $2n-1 \in O(n)$ nodes to estimate in the tree for m different values of the number of processors. The overall complexity of the algorithm is $O(nm^2)$. Moreover, the authors remark that if both $\mathcal{L}(l,j)-\mathcal{L}(l,j-1)$ and $\mathcal{L}(r,j)-\mathcal{L}(r,j-1)$ decrease when j increases, then $\mathcal{L}(S(l,r),j)$ has the same property. Using that property appropriately enables to obtain the optimal latency for chains (series graph) in $O(m\log n)$ by considering the whole chain at once (instead of using the Binary Decomposition Tree form). If a graph is only composed of parallel tasks, considering the whole graph at once leads to an algorithm of similar complexity. Finally, [CNNS94] shows how to efficiently solve the problem of optimizing the throughput under latency constraint.

[HM94] and its refinement [CHM95] are interested in optimizing the throughput of pipelined trees for database applications. Homogeneous processors are used, the communications do not overlap with computations, and the communications follow the bandwidth bounded multi-port model. Therefore the load of a processor is the sum of the weights of the nodes executed by this processor plus the weights of the edges to other processors. Since latency is not a concern here, there is no fine grain scheduling of tasks and data sets, but only a flow-like solution where each processor has a large buffer of data sets to execute. This solution is very similar to the notion of core schedule described in Section 4.2.2. The main contribution of [HM94] is the definition of a monotone tree, which is a modified version of a tree where two nodes linked by too high communication edge are merged. This technique is an extension of the monotonic chains used in [Iqb92]. It is shown that such a modification is

optimal.

[CHM95] presents two approximation algorithms for the previous problem. Both are based on a two-phase decomposition: first the tree is decomposed into a forest by removing some edges; then the trees are allocated to processors using LPT. Removing an edge incurs communication costs to both extremities of the edge. It is shown that if the obtained forest does not have too large trees and the load is kept reasonable, then LPT will generate an approximation of the optimal solution. Two tree decomposition algorithms follow. The first one is a simple greedy algorithm with approximation ratio 3.56, the second one is a more complex greedy algorithm with approximation ratio 2.87.

5.4 Scheduling with Replication

[SV95] addresses the problem of scheduling a chain of moldable tasks to optimize the throughput using replicated interval mappings: if a task is replicated, its whole interval is replicated too. The algorithm uses dynamic programming to find the intervals $I = \{I_1, \ldots, I_k\}$, and for interval $I_j \in I$, the number of processors m_{I_j} and the number of replications r_{I_j} for this interval, so that the period $\mathcal{P} = \max_{I_j \in I} \frac{p(I_j, m_{I_j})}{r_{I_j}}$ is minimized. Note that $p(I_j, m_{I_j})$ is the period of interval $I_j \in I$ executed on m_{I_j} processors. However, the algorithm does not return the periodic schedule that the system should follow. It just states where the tasks should be executed and it relies on a Demand Driven middleware to execute them correctly. A periodic schedule reaching the same throughput can be computed from the intervals, and the number of times they should be replicated. However, one needs to specify the execution of a number of data sets equal to the Least Common Multiple of the number of replication of all the intervals to completely provide the schedule. Indeed, if one task is replicated two times and another is replicated three times, the execution of six data sets must be unrolled for the schedule to be periodic.

[SV96] adds the computation of the latency to [SV95]. Since the graph is a chain and all the intervals are executed independently, it is possible to build a schedule that reaches the optimal latency for a given processor allocation $\mathcal{L} = \sum_{I_j \in I} p(I_j, m_{I_j})$. The interval that constrains the throughput must be executed without idle time, the preceding tasks are scheduled as late as possible and the following tasks are scheduled as soon as possible. The optimization of the latency under a throughput constraint is obtained using a dynamic programming algorithm, by forbidding the numbers of processors and numbers of replications for each interval that violate the throughput constraint.

5.5 General Method to Optimize the Throughput

[BHCF95] deals with executing a signal processing application on heterogeneous machines, where not all tasks can be executed on all type of processors. They schedule a precedence task graph of sequential monolithic tasks. Communications follow the bandwidth bounded multi-port model with latency, and they overlap with computations. The proposed algorithm relies on the notion of processor ordered schedule. (Recall that a schedule is processor ordered if the graph of the communications between the processors is acyclic). This property helps ensuring that precedence constraints are respected. The algorithm first builds a schedule by

clustering some tasks together to reduce the size of the graph. Then, an exponential algorithm finds the optimal processor ordered schedule of the clustered task graph. Finally, tasks are greedily moved from one processor to the other. The last two steps are alternatively executed as long as the throughput improves.

[Bey01] deals with scheduling pipelined task graphs on the grid. The resources of the grid are exploited using replication. The author proposes the Filter Copy Pipeline (FCP) algorithm. FCP considers the application graph in a topological order, and chooses the number of copies for each task, so that it can process the data it receives without getting a large backlog. In other words, if the predecessor of a task handles x data sets per time unit, FCP replicates this task to handle x data sets per time unit. Those replicas are allocated to processors using the earliest completion time rule. To improve this approach, [SFB+02] proposes Balanced Filter Copies that allocates a processor to a single task. In order to ensure the balance, it keeps track of the network bandwidth used while computing the schedule.

[TC99] is concerned with scheduling a pipelined task graph on a heterogeneous network of heterogeneous (related) processors with computation and communication overlap. Since the authors are only interested in the throughput, the problem reduces to a mapping of the tasks to the processors, and the throughput of the solution is given by the most loaded processor or communication link. The algorithm starts by ordering the tasks in depth-first traversal of a clustering tree. Then tasks are mapped to the processors using the following algorithm. The processors are, one after the other, loaded with the first unallocated tasks that minimize the maximum of three quantities: the current load, the perfectly balanced load of the unallocated tasks on the unallocated processors, and the communication volume between the tasks currently allocated to the processor and the unallocated tasks normalized by the total bandwidth of the processor. Finally, the obtained schedule is iteratively improved by unscheduling some of the tasks on the most loaded processors and/or links, and scheduling them again.

[YKS03] deals with scheduling arbitrary precedence task graphs on a Network of Workstations (NOW). The processors are heterogeneous (related) and allow for communication and computation overlap. The communications are modeled using the delay model where the delay is computed using a per link latency and bandwidth. Two objectives are optimized: the throughput and number of machines used from the NOW. The throughput is given by the user and then, the execution is cut in stages whose lengths are given by the throughput. A processor used in one stage is not reused in the next one, so that the throughput can be guaranteed. The tasks are allocated using the earliest task first heuristic. The authors also propose some techniques to compact the schedule, reducing the number of processors used.

5.6 General Method to Optimize Throughput and Latency

[GRRL05] is interested in scheduling a pipelined precedence task graph on a homogeneous cluster with communication and computation overlap to optimize both latency and throughput. The network is assumed to be completely connected and the delay model is used. The delay between two tasks scheduled on two processors is computed using a latency plus bandwidth model. The EXPERT algorithm optimizes the latency under a throughput constraint. Given a throughput goal $\mathcal{T} = 1/\mathcal{P}$, all tasks are partitioned into stages. Each stage is identified by an in-

teger. Task t is allocated to the minimum stage k such that $topLevel(t) \leq k \times \mathcal{P}$, where $topLevel(t)$ is the critical path from the root of the graph to the completion of t. Then, EXPERT considers all the paths of the graph from root to sink in decreasing order of length, including communication delays, and for each edge of each path it applies the following greedy rule: if the two clusters linked by the edge belong to the same stage, and if the sum of the processing times of the tasks in these clusters is smaller than $\mathcal{P}$, then the two clusters are merged. Finally, inter-stage clusters are merged as long as the sum of the processing times of the tasks of the resulting cluster is less than $\mathcal{P}$. Each cluster is assigned to a different processor (and the throughput goal is declared infeasible if there are not enough processors). Communication between the processors are grouped at the end of the execution of the cluster they are assigned to.

[HO99] deals with arbitrary application graphs and homogeneous processors and network links. The technique was originally designed for hypercube networks, but can be adapted to arbitrary networks. It assumes communication and computation overlap. The authors are interested in optimizing both latency and throughput. The proposed algorithm only provides a processor allocation, and the periodic schedule is reconstructed using a technique similar to the one presented in Section 4.2.5: within a period, the tasks are ordered in topological order. If a task precedence constraint is not satisfied inside the current period, it enforces the dependency from the previous period. Given a target period, and therefore throughput, the proposed method has three phases. The first phase groups the tasks in as many clusters as there are processors. This phase orders communications in non-increasing order of their size, and greedily considers grouping the tasks at both ends of the communication inside the same cluster, unless both tasks are already assigned to a cluster, or the assignment would make the sum of the processing times of the tasks in the cluster larger than the period. At the end of this phase, tasks that are not assigned to a cluster are assigned using a first fit rule. Then, in the second phase, the clusters are mapped to computation nodes to minimize the amount of communication. This is done by first mapping the clusters randomly to the nodes. Then the processor set is cut in two equals parts and clusters are pair-wise exchanged to decrease communications on the processor cut. The communications in each part of the processor set are optimized by applying the "cut in two parts and exchange clusters" procedure recursively. Finally, in the third phase, the solution is improved iteratively by moving tasks between processors to decrease the load of the most loaded link.

[VÇK+10] considers optimizing the latency and throughput of arbitrary DAGs on homogeneous processors linked by a network of different bandwidths, with communication/computation overlap, and using replication and duplication. Communications are explicitly scheduled according to the k-port model. The algorithm operates in three phases and takes a throughput constraint as a parameter. The first phase groups tasks together in as many clusters as necessary to match the throughput constraint. This is achieved by considering the replication of each task to deal with computational bottlenecks, and the duplication of each task and merging the clusters of the tasks at both ends of an edge to decrease communication bottlenecks. For each alteration of the mapping considered, the throughput is

evaluated by scheduling the computations and the communications using a greedy algorithm. In a second phase, the number of clusters is reduced to the number of processors in the system by merging clusters to minimize processor idle times. Finally, in the last phase, the latency is minimized by considering for each task of the critical path its duplication, and merging to its predecessor cluster or successor cluster.

6. CONCLUSION AND FUTURE WORK

In this survey, we have presented an overview of pipelined workflow scheduling, a problem that asks for an efficient execution of a streaming application that operates on a set of consecutive data sets. We described the components of application and platform models, and how a scheduling problem can be formulated for a given application. We presented a brief summary of the solution methods for specific problems, highlighting the frontier between polynomial and NP-hard optimization problems.

Although there is a significant body of literature for this complex problem, realistic application scenarios still call for more work in the area, both theoretical and practical. When developing solutions for real-life applications, one has to consider all the ingredients of the schedule as a whole, including detailed communication models and memory requirements (especially when more than one data set is processed in a single period). Such additional constraints make the development of efficient scheduling methods even more difficult.

As the literature shows, having structure either in the application graph or in the execution platform graph dramatically helps for deriving effective solutions. We think that extending this concept to the schedule could be useful too. For example, for scheduling arbitrary DAGs, developing structured schedules, such as convex clustered schedules, has a potential for yielding new results in this area.

Finally, as the domain evolves, new optimization criteria must be introduced. In this paper, we have mainly dealt with throughput and latency. Other performance-related objectives arise with the advent of very large-scale platforms, such as increasing the reliability of the schedule (e.g., through task duplication). Environmental and economic criteria, such as the energy dissipated throughout the execution, or the rental cost of the platform, are also likely to play an increasing role. Altogether, we believe that future research will be devoted to optimizing several performance-oriented and environmental criteria simultaneously. Achieving a reasonable trade-off between all these multiple and antagonistic objectives will prove a very interesting algorithmic challenge.

ACKNOWLEDGMENTS

We would like to wholeheartedly thank the three reviewers, whose comments and suggestions greatly helped us to improve the final version of the paper.

This work was supported in parts by the DOE grant DE-FC02-06ER2775; by AFRL/DAGSI Ohio Student-Faculty Research Fellowship RY6-OSU-08-3; by the NSF grants CNS-0643969, OCI-0904809, and OCI-0904802, and by the French ANR StochaGrid and RESCUE projects. Anne Benoit and Yves Robert are with the Institut Universitaire de France.

REFERENCES

[ABMR10] Kunal Agrawal, Anne Benoit, Loic Magnan, and Yves Robert. Scheduling algorithms for linear workflow optimization. In *IPDPS'2010, the 24th IEEE International Parallel and Distributed Processing Symposium*, Atlanta, USA, 2010. IEEE Computer Society Press.

[ABR08] Kunal Agrawal, Anne Benoit, and Yves Robert. Mapping linear workflows with computation/communication overlap. In *ICPADS'2008, the 14th IEEE International Conference on Parallel and Distributed Systems*, Melbourne, Australia, December 2008. IEEE.

[AISS95] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One step closer towards a realistic model for parallel computation. In *SPAA '95: Proceedings of the seventh annual symposium on Parallelism in algorithms and architectures*, 1995.

[AJLA95] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27:367–432, 1995.

[AK98] Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):872–892, September 1998.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS'1967, the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, April 1967. ACM.

[AMPP04] Alessandro Agnetis, Pitu B. Mirchandani, Dario Pacciarelli, and Andrea Pacifici. Scheduling problems with two competing agents. *Operations Research*, 52(2):229–242, April 2004.

[BBG+09] Xavier Besseron, Slim Bouguerra, Thierry Gautier, Erik Saule, and Denis Trystram. Fault tolerance and availability awareness in computational grids. In F. Magoules, editor, *Fundamentals of Grid Computing*, Numerical Analysis and Scientific Computing. Chapman and Hall/CRC Press, December 2009. ISBN: 978-1439803677.

[BCC+01] Francine Berman, Andrew Chien, Keith Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, John Mellor-Crumme, Dan Reed, Linda Torczon, and Rich Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High Performance Computing Applications*, 15(4):327–344, Winter 2001.

[Ben09] Anne Benoit. Scheduling pipelined applications: models, algorithms and complexity, July 2009. Habilitation à diriger des recherches, École normale supérieure de Lyon.

[Bey01] Michael D. Beynon. *Supporting Data Intensive Applications in a Heterogeneous Environment*. PhD thesis, University of Maryland, 2001.

[BGGR09] Anne Benoit, Bruno Gaujal, Matthieu Gallet, and Yves Robert. Computing the throughput of replicated workflows on heterogeneous platforms. In *ICPP'2009, the 38th International Conference on Parallel Processing*, Vienna, Austria, 2009. IEEE Computer Society Press.

[BHCF95] Sati Banerjee, Takeo Hamada, Paul M. Chau, and Ronald D. Fellman. Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems. *IEEE Transactions on Signal Processing*, 43(6):1468–1484, June 1995.

[BHS+94] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *Journal on Parallel and Distributed Computing*, 21:4–14, 1994.

[BKÇ+01] Michael D. Beynon, Tahsin Kurc, Ümit V. Çatalyürek, Chialin Chang, Alan Sussman, and Joel Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, October 2001.

[BKP07] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Speed scaling to manage energy and temperature. *Journal of the ACM*, 54(1):1 – 39, 2007.

[BKRSR09] Anne Benoit, Harald Kosch, Veronika Rehn-Sonigo, and Yves Robert. Multicriteria Scheduling of Pipeline Workflows (and Application to the JPEG Encoder). *International Journal of High Performance Computing Applications*, 2009.

[BLMR04] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Assessing the impact and limits of steady-state scheduling for mixed task and data parallelism on heterogeneous platforms. In *ISPDC'04, the 3rd International Symposium on Parallel and Distributed Computing/3rd International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 296–302, Washington, DC, USA, 2004. IEEE Computer Society.

[BML+06] Shawn Bowers, Timothy M. McPhillips, Bertram Ludäscher, Shirley Cohen, and Susan B. Davidson. A model for user-oriented data provenance in pipelined scientific workflows. In *Provenance and Annotation of Data, International Provenance and Annotation Workshop (IPAW)*, pages 133–147, 2006.

[Bok88] Shahid H. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. *IEEE Transactions on Computers*, 37(1):48–57, 1988.

[BR08] Anne Benoit and Yves Robert. Mapping pipeline skeletons onto heterogeneous platforms. *Journal on Parallel and Distributed Computing*, 68(6):790–808, 2008.

[BR09] Anne Benoit and Yves Robert. Multi-criteria mapping techniques for pipeline workflows on heterogeneous platforms. In George A. Gravvanis, John P. Morrison, Hamid R. Arabnia, and David A. Power, editors, *Recent developments in Grid Technology and Applications*, pages 65–99. Nova Science Publishers, 2009.

[BR10] Anne Benoit and Yves Robert. Complexity results for throughput and latency optimization of replicated and data-parallel workflows. *Algorithmica*, 57(4):689–724, August 2010.

[BRGR10] Anne Benoit, Paul Renaud-Goud, and Yves Robert. Performance and energy optimization of concurrent pipelined applications. In *IPDPS'2010, the 24th IEEE International Parallel and Distributed Processing Symposium*, Atlanta, USA, 2010. IEEE Computer Society Press.

[BRP03] Prashanth B. Bhat, C.S. Raghavendra, and Viktor K. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal on Parallel and Distributed Computing*, 63(3):251–263, March 2003.

[BRSR07] Anne Benoit, Veronika Rehn-Sonigo, and Yves Robert. Multi-criteria scheduling of pipeline workflows. In *HeteroPar'07, the 6th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Austin, Texas, USA, June 2007.

[BRSR08] Anne Benoit, Veronika Rehn-Sonigo, and Yves Robert. Optimizing latency and reliability of pipeline workflow applications. In *HCW'08, the 17th International Heterogeneity in Computing Workshop*, Miami, USA, April 2008. IEEE.

[BRT09] Anne Benoit, Yves Robert, and Eric Thierry. On the complexity of mapping linear chain applications onto heterogeneous platforms. *Parallel Processing Letters*, 19(3):383–397, March 2009.

[Bru07] Peter Brucker. *Scheduling Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, fifth edition, 2007.

[BS05] Ragnhild Blikberg and Tor Sørevik. Load balancing and OpenMP implementation of nested parallelism. *Parallel Computing*, 31:984–998, 2005.

[CHM95] Chandra Chekuri, Waqar Hasan, and Rajeev Motwani. Scheduling problems in parallel query optimization. In *PODS'1995, the 14th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 255–265, New York, NY, USA, 1995. ACM Press.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *PPOPP'1993, the 4th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, New York, NY, USA, 1993. ACM.

[CLW+00] Alok Choudhary, Wei-keng Liao, Donald Weiner, Pramod Varshney, Richard Linderman, Mark Linderman, and Russell Brown. Design, implementation and evaluation of parallel pipelined STAP on parallel computers. *IEEE Transactions on Aerospace and Electronic Systems*, 36(2):655 – 662, April 2000.

[CNNS94] Alok Choudhary, Bhagirath Narahari, David Nicol, and Rahul Simha. Optimal processor assignment for a class of pipeline computations. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):439–443, April 1994.

[Col04] Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[Dav78] Alan L. Davis. Data driven nets: A maximally concurrent, procedural, parallel process representation for distributed control systems. Technical report, Technical Report, Department of Computer Science, University of Utah, Salt Lake City, Utah, 1978.

[DBGK03] Ewa Deelman, James Blythe, Yolanda Gil, and Carl Kesselman. *Grid Resource Management*, chapter Workflow management in GriPhyN. Springer, 2003.

[Den74] Jack B. Dennis. First version of a data flow procedure language. In *Symposium on Programming*, pages 362–376, 1974.

[Den80] Jack B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980.

[Dev09] UmaMaheswari C. Devi. Scheduling recurrent precedence-constrained task graphs on a symmetric shared-memory multiprocessor. In Springer, editor, *Euro-Par 2009 Parallel Processing*, pages 265–280, August 2009.

[dNFJG05] Luiz Thomaz do Nascimento, Renato A. Ferreira, Wagner Meira Jr., and Dorgival Guedes. Scheduling data flow applications using linear programming. In *ICPP'2005, the 34th International Conference on Parallel Processing*, pages 638–645, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[DRST09] Pierre-Francois Dutot, Krzyztof Rzadca, Erik Saule, and Denis Trystram. Multi-objective scheduling. In Yves Robert and Frederic Vivien, editors, *Introduction to Scheduling*. CRC Press, November 2009.

[DRV00] Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization*. Birkhauser, 2000.

[DShS+05] Ewa Deelman, Gurmeet Singh, Mei hui Su, James Blythe, A Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13:219–237, 2005.

[FJP+05] Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto, Jr., and Hong-Linh Truong. ASKALON: a tool set for cluster and Grid computing: Research Articles. *Concurrency and Computation: Practice and Experience*, 17:143–169, February 2005.

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[FPT04] Thomas Fahringer, Sabri Pllana, and Johannes Testori. Teuta: Tool support for performance modeling of distributed and parallel applications. In *International Conference on Computational Science, Tools for Program Development and Analysis in Computational Science*, Krakow, Poland, Jun 2004.

[FRS+97] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, LNCS, pages 1–34. Springer, 1997.

[GIS77] Teofilo F. Gonzalez, Oscar H. Ibarra, and Sartaj Sahni. Bounds for LPT schedules on uniform processors. *SIAM Journal on Computing*, 6:155–166, 1977.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Freeman, San Francisco, 1979.

[GMS04] Frederic Guinand, Aziz Moukrim, and Eric Sanlaville. Sensitivity analysis of tree scheduling on two machines with communication delays. *Parallel Computing*, 30:103–120, 2004.

[GMW05] Martin Gairing, Burkhard Monien, and Andreas Woclaw. *Automata, Languages and Programming*, volume 3580, chapter A Faster Combinatorial Approximation Algorithm for Scheduling Unrelated Parallel Machines, pages 828–839. Springer, aug 2005.

[Gra66] Ronald L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[Gra69] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[GRR+06] Fernando Guirado, Ana Ripoll, Concepció Roig, Aura Hernandez, and Emilio Luque. Exploiting throughput for pipeline execution in streaming image processing applications. In *Euro-Par 2006, Parallel Processing*, LNCS 4128, pages 1095–1105. Springer, 2006.

[GRRL05] Fernando Guirado, Ana Ripoll, Concepcio Roig, and Emilio Luque. Optimizing latency under throughput requirements for streaming applications on cluster execution. In *Cluster Computing, 2005. IEEE International*, pages 1–10, September 2005.

[GST09] Alain Girault, Erik Saule, and Denis Trystram. Reliability versus performance for critical applications. *Journal on Parallel and Distributed Computing*, 69(3):326–336, 2009.

[HÇ09] Timothy D. R. Hartley and Ümit V. Çatalyürek. A Component-Based Framework for the Cell Broadband Engine. In *Proceedings of 23rd International. Parallel and Distributed Processing Symposium, The 18th Heterogeneous Computing Workshop (HCW 2009)*, May 2009.

[HÇR+08] Timothy D. R. Hartley, Ümit V. Çatalyürek, Antonio Ruiz, Francisco Igual, Rafael Mayo, and Manuel Ujaldon. Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores. In *Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008*, pages 15–25, 2008.

[HFB+09] Timothy D. R. Hartley, Ahmed R. Fasih, Charles A. Berdanier, Fusun Ozguner, and Ümit V. Çatalyürek. Investigating the Use of GPU-Accelerated Nodes for SAR Image Formation. In *Proceedings of the IEEE International Conference on Cluster Computing, Workshop on Parallel Programming on Accelerator Clusters (PPAC)*, 2009.

[HL97] Soonhoi Ha and Edward A. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers*, 46:768–778, July 1997.

[HM94] Waqar Hasan and Rajeev Motwani. Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. In *VLDB*, pages 36–47, 1994.

[HNC92] Yijie Han, Bhagirath Narahari, and Hyeong-Ah Choi. Mapping a chain task to chained processors. *Information Processing Letters*, 44:141–148, 1992.

[HO99] Stephen L. Hary and Fusun Ozguner. Precedence-constrained task allocation onto point-to-point networks for pipelined execution. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):838–851, 1999.

[Hoc97] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-hard problems*. PWS publishing company, 1997.

[HP03] Bo Hong and Viktor K. Prasanna. Bandwidth-aware resource allocation for heterogeneous computing systems to maximize throughput. In *ICPP'2003, the 32th International Conference on Parallel Processing*. IEEE Computer Society Press, 2003.

[HS87] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems: Practical and theoretical results. *Journal of ACM*, 34:144–162, 1987.

[HS88] Dorit S. Hochbaum and David B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM Journal on Computing*, 17(3):539 – 551, 1988.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys'2007, the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[Iqb92] Mohammad Ashraf Iqbal. Approximate algorithms for partitioning problems. *International Journal of Parallel Programming*, 20(5):341–361, 1992.

[JPG04] Ravindra Jejurikar, Cristiano Pereira, and Rajesh Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of DAC'04, the 41st annual Design Automation Conferencea*, pages 275–280, New York, NY, USA, 2004. ACM.

[JV96] Jon Jonsson and Jonas Vasell. Real-time scheduling for pipelined execution of data flow graphs on a realistic multiprocessor architecture. In *ICASSP-96: Proceedings of the 1996 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 6, pages 3314–3317, 1996.

[KA99a] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, December 1999.

[KA99b] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[KA02] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann, 2002.

[Kah74] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[KGS04] Jihie Kim, Yolanda Gil, and Marc Spraragen. A knowledge-based approach to interactive workflow composition. In *14th International Conference on Automatic Planning and Scheduling (ICAPS 04)*, 2004.

[KN10] Ekasit Kijsipongse and Sudsanguan Ngamsuriyaroj. Placing pipeline stages on a Grid: Single path and multipath pipeline execution. *Future Generation Computer Systems*, 26(1):50 – 62, 2010.

[KRC+99] Kathleen Knobe, James M. Rehg, Arun Chauhan, Rishiyur S. Nikhil, and Umakishore Ramachandran. Scheduling constrained dynamic applications on clusters. In *Supercomputing'1999, the 1999 ACM/IEEE conference on Supercomputing*, page 46, New York, NY, USA, 1999. ACM.

[LKdPC10] Eugene Levner, Vladimir Kats, David Alcaide Lopez de Pablo, and T.C.E. Cheng. Complexity of cyclic scheduling problems: A state-of-the-art survey. *Computers and Industrial Engineering*, 59:352–361, 2010.

[LLM88] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor-a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104 –111, jun 1988.

[LLP98] Myungho Lee, Wenheng Liu, and Viktor K. Prasanna. A mapping methodology for designing software task pipelines for embedded signal processing. In *Proceedings of the Workshop on Embedded HPC Systems and Applications of IPPS/SPDP*, pages 937–944, 1998.

[LP95] Edward A. Lee, , and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773 –801, may 1995.

[LST90] Jan K. Lenstra, David B. Shmoys, and Eva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[LT02] Renaud Lepere and Denis Trystram. A new clustering algorithm for large communication delays. In *International Parallel and Distributed Processing Symposium (IPDPS'2002)*. IEEE Computer Society Press, 2002.

[MCG+08] Andreu Moreno, Eduardo César, Alex Guevara, Joan Sorribes, Tomas Margalef, and Emilio Luque. Dynamic Pipeline Mapping (DPM). In Springer, editor, *Euro-Par 2008 Parallel Processing*, pages 295–304, August 2008. It looks like there is a journal vesion of it now : http://www.sciencedirect.com/science/article/pii/S0167819111001566.

[MGPD+08] Allan MacKenzie-Graham, Arash Payan, Ivo D. Dinov, John D. Van Horn, and Arthur W. Toga. Neuroimaging Data Provenance Using the LONI Pipeline Workflow Environment. In *Provenance and Annotation of Data, International Provenance and Annotation Workshop (IPAW)*, pages 208–220, 2008.

[Mic09] Microsoft. AXUM webpage. http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx, 2009.

[Mil99] Mark P. Mills. *The internet begins with coal: A preliminary exploration of the impact of the Internet on electricity consumption: a green policy paper for the Greening Earth Society*. Mills-McCarthy & Associates, 1999.

[MO95] Fredrik Manne and Bjørn Olstad. Efficient partitioning of sequences. *IEEE Transactions on Computers*, 44(11):1322–1326, 1995.

[Nic94] David Nicol. Rectilinear partitioning of irregular data parallel computations. *Journal on Parallel and Distributed Computing*, 23:119–134, 1994.

[NTS+08] Hristo Nikolov, Mark Thompson, Todor Stefanov, Andy D. Pimentel, Simon Polstra, R. Bose, Claudiu Zissulescu, and Ed F. Deprettere. Daedalus: toward composable multimedia MP-SoC design. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 574–579, New York, NY, USA, 2008. ACM.

[OGA+06] Thomas Oinn, Mark Greenwood, Matthew Addis, Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Christopher Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, August 2006.

[OYI01] Takanori Okuma, Hiroto Yasuura, and Tohru Ishihara. Software energy reduction techniques for variable-voltage processors. *Design Test of Computers, IEEE*, 18(2):31 –41, March 2001.

[PA04] Ali Pınar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *Journal on Parallel and Distributed Computing*, 64(8):974–996, 2004.

[Pra04] Rajesh B. Prathipati. Energy efficient scheduling techniques for real-time embedded systems. Master's thesis, Texas A&M University, May 2004.

[PST05] Jonathan E. Pecero-Sanchez and Denis Trystram. A new genetic convex clustering algorithm for parallel time minimization with large communication delays. In Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, Oscar G. Plata, P. Tirado, and Emilio L. Zapata, editors, *PARCO*, volume 33 of *John von Neumann Institute for Computing Series*, pages 709–716. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[PTA08] Ali Pınar, E. Kartal Tabak, and Cevdet Aykanat. One-dimensional partitioning for heterogeneous systems: Theory and practice. *Journal on Parallel and Distributed Computing*, 68:1473–1486, 2008.

[PY00] Christos H. Papadimitriou and Mihalis Yannakakis. On the approximability of trade-offs and optimal access of web sources. In FOCS, editor, *41st Annual Symposium on Foundations of Computer Science*, pages 86–92, 2000.

[RA01] Samantha Ranaweera and Dharma P. Agrawal. Scheduling of periodic time critical applications for pipelined execution on heterogeneous systems. In *ICPP '02: Proceedings of the 2001 International Conference on Parallel Processing*, pages 131–140, Washington, DC, USA, 2001. IEEE Computer Society.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O' Reilly, 2007.

[RKO+03] Anthony Rowe, Dimitrios Kalaitzopoulos, Michelle Osmond, Moustafa Ghanem, and Yike Guo. The discovery net system for high throughput bioinformatics. *Bioinformatics*, 19(Suppl 1):i225–31, 2003.

[RS87] Vic J. Rayward-Smith. UET scheduling with interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.

[RSBJ95] Vic J. Rayward-Smith, F. Warren Burton, and Gareth J. Janacek. Scheduling parallel program assuming preallocation. In P. Chrétienne, E. G. Coffman Jr., J. K. Lenstra, and Z. Liu, editors, *Scheduling Theory and its Applications*, pages 146–165. Wiley, 1995.

[SFB+02] Matthew Spencer, Renato Ferreira, Michael D. Beynon, Tahsin Kurc, Ümit V. Çatalyürek, Alan Sussman, and Joel Saltz. Executing multiple pipelined data analysis operations in the grid. In *Supercomputing'2002, the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[SKS+09] Olcay Sertel, Jun Kong, Hiroyuki Shimada, Ümit V. Çatalyürek, Joel H. Saltz, and Metin N. Gurcan. Computer-aided prognosis of neuroblastoma on whole-slide images: Classification of stromal development. *Pattern Recognition*, 42(6):1093–1103, 2009.

[SP04] Taher Saif and Manish Parashar. Understanding the behavior and performance of non-blocking communications in MPI. In *Euro-Par 2004 Parallel Processing*, LNCS 3149, pages 173–182. Springer, 2004.

[SRM06] Vivy Suhendra, Chandrashekar Raghavan, and Tulika Mitra. Integrated scratch-pad memory optimization and task scheduling for MPSoC architectures. In *CASES '06: ACM/IEEE International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, October 2006.

[SV95] Jaspal Subhlok and Gary Vondran. Optimal mapping of sequences of data parallel tasks. In *PPOPP'1995, the 5th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 134–143, New York, NY, USA, 1995. ACM.

[SV96] Jaspal Subhlok and Gary Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *SPAA'1996, the 8th annual ACM symposium on Parallel algorithms and architectures*, pages 62–71, New York, NY, USA, 1996. ACM.

[TB07] Vincent T'kindt and Jean-Charles Billaut. *Multicriteria Scheduling*. Springer, 2007.

[TBE07] Vincent T'kindt, K. Bouibede-Hocine, and Carl Esswein. Counting and enumeration complexity with application to multicriteria scheduling. *Annals of Operations Research*, April 2007.

[TC99] Kenjiro Taura and Andrew Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *HCW'1999, the Heterogeneous Computing Workshop*, pages 102–115. IEEE Computer Society Press, 1999.

[TFG+08] George Teodoro, Daniel Fireman, Dorgival Guedes, Wagner Meira Jr., and Renato Ferreira. Achieving multi-level parallelism in filter-labeled stream programming model. In *ICPP'2008, the 37th International Conference on Parallel Processing*, 2008.

[TTL02] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the Grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[TWML01] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[TWS03] Valerie Taylor, Xingfu Wu, and Rick Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Performance Evaluation Review*, 30:13–18, March 2003.

[VÇK+07] Nagavijayalakshmi Vydyanathan, Ümit V. Çatalyürek, Tahsin M. Kurc, P. Sadayappan, and Joel H. Saltz. Toward optimizing latency under throughput constraints for application workflows on clusters. In *Euro-Par 2007 Parallel Processing*, pages 173–183, 2007.

[VÇK+10] Nagavijayalakshmi Vydyanathan, Ümit V. Çatalyürek, Tahsin M. Kurc, P. Sadayappan, and Joel H. Saltz. Optimizing latency and throughput of application workflows on clusters. *Parallel Computing*, In Press, 2010.

[VTL82] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. *SIAM Journal on Computing*, 11(2):298–313, 1982.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge MA, 1989.

[WSH99] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Gener. Comput. Syst.*, 15:757–768, October 1999.

[WvLDW10] Lizhe Wang, G. von Laszewski, J. Dayal, and Fugang Wang. Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS. In *Proceedings of CCGrid'2010, the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 368 –377, May 2010.

[YB05] Jia Yu and Rajkumar Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 3:171–200, 2005. 10.1007/s10723-005-9010-8.

[YDS95] Frances Yao, Alan Demers, and Scott Shenker. A scheduling model for reduced CPU energy. In *Proceedings of FOCS '95, the 36th Annual Symposium on Foundations of Computer Science*, page 374, Washington, DC, USA, 1995. IEEE Computer Society.

[YKS03] Mau-Tsuen Yang, Rangachar Kasturi, and Anand Sivasubramaniam. A Pipeline-Based Approach for Scheduling Video Processing Algorithms on NOW. *IEEE Transactions on Parallel and Distributed Systems*, 14(2):119–130, 2003.